Think BIGGER

DEEP DIVE FÜR FACHLEUTE

Snowflake

Cloud Data Warehouse für analytische Workloads und elastische Skalierung

Autor	Erhard Rainer
Veröffentlicht	30.04.2016
Aktualisiert	06.07.2026
Umfang	ca. 37 Min. Lesezeit
Online	www.thinkbigger.at/wissen/snowflake/

thinkbigger.at

Business Intelligence · Machine Learning · KI-Agenten · Schulungen

Erhard Rainer · Schönaugasse 106 · 8020 Graz · info@think-bigger.at

Inhalt

0. Überblick (für Eilige) 2
1. Einordnung: Wann Snowflake die richtige Wahl ist 4
2. Architektur & Kernkonzepte 7
3. Datenmodellierung in der Praxis 11
4. SQL, Laden & Transformieren 14
5. Datenprodukte, Zusammenarbeit & Resilienz 18
6. Betrieb, Monitoring & Troubleshooting 20
7. Performance, Skalierung & Kostenkontrolle 23
8. Sicherheit, Governance & Compliance 26
9. Weiterführende Links + Uebungsaufgaben 29

0. Überblick (für Eilige)

Snowflake ist eine cloud-native Datenplattform für analytische SQL-Workloads, ELT, Datenfreigabe, Governance und zunehmend auch AI/ML-nahe Datenprodukte. Die Plattform ist besonders stark, wenn mehrere Teams dieselben Daten nutzen sollen, aber nicht dieselben Compute-Ressourcen, Betriebsprozesse und Lastprofile teilen wollen. Diese Seite ist eine praxisorientierte Referenz für Architektur, Datenmodellierung, Laden und Transformieren, Betrieb, Kostenkontrolle und Sicherheit. Ziel ist nicht nur "was ist Snowflake", sondern "wie baue und betreibe ich Snowflake so, dass Performance, Governance und Kosten in Produktion vorhersagbar bleiben".

> Hinweis zum Versionsstand: Diese Referenz ist fachlich auf Stand **11. April 2026**. Snowflake ist ein SaaS-Dienst mit Continuous Delivery und ohne klassisches Produkt-Release im Sinne einer fest installierten Versionslinie. In den offiziellen Release Notes waren zu diesem Zeitpunkt die juengste abgeschlossene Server-Linie `10.12` (3.-8. April 2026) sowie die Vorschau `10.13` sichtbar. Editions- und Preview-sensitive Aussagen sind im Text entsprechend markiert.

0.1. Inhaltsverzeichnis (kurz)

- 0. Überblick (für Eilige)
- 1. Einordnung: Wann Snowflake die richtige Wahl ist
- 2. Architektur & Kernkonzepte
- 3. Datenmodellierung in der Praxis
- 4. SQL, Laden & Transformieren
- 5. Datenprodukte, Zusammenarbeit & Resilienz
- 6. Betrieb, Monitoring & Troubleshooting
- 7. Performance, Skalierung & Kostenkontrolle
- 8. Sicherheit, Governance & Compliance
- 9. Weiterführende Links + Uebungsaufgaben

0.2. Quick-Glossar

- **Virtual Warehouse**: Unabhängiger Compute-Cluster für SQL, Snowpark und bestimmte Workloads.
- **Micro-Partition**: Automatisch erzeugte, kleine Speichereinheit einer Tabelle; Grundlage für Pruning und Clustering.
- **Pruning**: Laufzeitmechanismus, der irrelevante Micro-Partitions und Spaltenbereiche von Scans ausschliesst.
- **Clustering Key**: Zusatzausrichtung für grosse Tabellen, um Pruning bei bestimmten Zugriffspfaden zu verbessern.
- **Stage**: Ablage für Lade- und Entladeprozesse, intern oder extern.
- **Snowpipe**: Serverless-Micro-Batch-Laden aus Stages.
- **Stream**: CDC-Objekt, das Änderungen einer Quelle über einen Offset sichtbar macht.
- **Task**: Geplanter oder ereignisgesteuerter Ausfuehrungspunkt für SQL/Pipelines.
- **Dynamic Table**: Tabelle mit automatischer Aktualisierung auf Basis einer Query und eines Freshness-Ziels.
- **Time Travel**: Rückgriff auf historische Daten-/Objektzustaende für Abfragen, Restore und Klonen.

- **Fail-safe**: Von Snowflake verwaltete Notfall-Wiederherstellung historischer Daten nach Ende der Time-Travel-Phase.
- **Secure Data Sharing**: Teilen von Daten zwischen Accounts ohne Kopie der eigentlichen Daten.
- **Search Optimization Service**: Zusatzservice für selektive Punkt- und Teilmengenabfragen.
- **QAS**: Query Acceleration Service; beschleunigt bestimmte grosse Scan-/Filter-Workloads über gemeinsame Compute-Ressourcen.

0.3. Lernpfad: Vom Einstieg zur Produktion

1. Verstehe zuerst das Betriebsmodell: Storage, Compute, Cloud Services, Warehouses, serverless Zusatzdienste.
2. Modelliere danach nicht nur Tabellen, sondern Datenfluesse, Verantwortlichkeiten und Kostenpfade.
3. Baue dann Lade- und Transformationsmuster bewusst: `COPY INTO`, Snowpipe, Streams/Tasks oder Dynamic Tables.
4. Miss anschliessend reale Query-Pfade, Queueing, Cache-Nutzung und Credit-Verbrauch.
5. Sichere zuletzt das Setup ab: Rollenmodell, Masking, Row Access, Budgets, Resilienz und Freigabeprozesse.

0.4. Mini-Projekt (durchgehend): SaaS-Analytics-Plattform mit Near-Real-Time-Faehigkeit

Dieses Mini-Projekt begleitet alle Kapitel. **Szenario:**

- Ein SaaS-Anbieter verarbeitet Produkt-Events, Billing-Daten und Support-Tickets.
- Datenzufluss: täglich ca. 2 TB Batch-Dateien plus kontinuierliche App-Events. **Nutzergruppen:**
- BI-Team für Dashboards.
- Finance für tägliche Abschluesse.
- Data Science für Feature Engineering.
- Externe Partner für kontrollierte Datenfreigabe. **Anforderungen:**
- Operative Dashboards mit maximal 5 Minuten Freshness.
- Monatsabschlussdaten mit klar nachvollziehbarem Datenstand.
- Rollenbasierte Sichtbarkeit für PII und regionale Daten.
- Vorhersagbare Kosten für ETL, BI und experimentelle Analysen.

Wir verwenden dieses Szenario fortlaufend für:

- Wahl der Tabellen- und Layer-Strategie.
- Warehouse-Isolierung nach Workload.
- Auswahl von Streams/Tasks vs. Dynamic Tables.
- Freigabe, Governance und Kostenkontrolle.

0.5. Profi-Heuristiken

- Snowflake ist besonders stark, wenn viele Workloads dieselben Daten nutzen, aber nicht dieselbe Rechenkapazitaet teilen sollen.
- Mehr Warehouse-Groesse heilt kein schlechtes Pruning.
- Clustering Keys sind ein Skalpell, kein Standardpflaster für jede Tabelle.
- Serverless Features senken Betriebsaufwand, können aber Kostenpfade unsichtbarer machen.

- Persisted Results und Warehouse-Cache sind hilfreich, aber kein Ersatz für sauberes Datenmodell und Query-Disziplin.
- Sharing ohne Kopie ist ein Governance-Vorteil, kein Freifahrtschein für unklare Verantwortungen.
- Time Travel ist keine Backup-Strategie für jedes Szenario, sondern Teil eines größeren Resilienzmodells.

0.6. So setzt du es um - Checkliste

- Trenne mindestens `etl`, `bi`, `data_science` und `admin` gedanklich als unterschiedliche Workloads.
- Lege pro Workload ein Warehouse-, Kosten- und Ownership-Modell fest.
- Entscheide bewusst, welche Tabellen `permanent`, `transient`, `temporary` oder `hybrid` sein sollen.
- Dokumentiere pro Datenprodukt Freshness-Ziel, Sicherheitsniveau und Freigabemodell.
- Miss früh Pruning, Queueing, Cache-Nutzung und serverless Nebenkosten.

0.7. Mentales Modell für den Alltag

Wenn du Snowflake sicher in Projekten einsetzen willst, hilft ein klares mentales Modell: Snowflake ist nicht nur "ein Cloud Data Warehouse", sondern eine Kombination aus Datenplattform, Betriebsmodell und Kostenmotor; genau deshalb entstehen die größten Fehlentscheidungen nicht bei der Syntax eines `SELECT`, sondern dort, wo Teams Datenmodell, Warehouse-Design und Governance voneinander trennen. Viele Einführungen scheitern daran, dass Snowflake wie ein unsichtbar skalierender SQL-Dienst behandelt wird: Data Engineering laedt Daten hinein, BI freut sich über schnelle erste Abfragen, Finance sieht später die Rechnung, Security versucht nachträglich Masking zu ergänzen, und niemand fuehlt sich für die Wechselwirkungen verantwortlich. In der Praxis ist Snowflake jedoch vor allem ein System expliziter Verträge: Welche Datenprodukte existieren? Wie frisch müssen sie sein? Welche Rollen dürfen was sehen? Welcher Workload darf welchen Warehouse-Typ nutzen? Welche Features sind user-managed, welche serverless, welche editionsabhaengig? Sobald diese Verträge fehlen, wird Snowflake zwar technisch weiter funktionieren, aber organisatorisch unberechenbar: Dashboards geraten in Konkurrenz zu ETL-Laeufen, Dynamic Tables refreshen zwar "automatisch", aber ohne Kostenleitplanken, Materialized Views bringen Performance, aber keiner fuehlt sich für deren Wartungskosten zuständig, und Secure Data Sharing wird aktiviert, ohne dass das fachliche Datenprodukt sauber definiert ist. Das alltagstaugliche Leitbild lautet deshalb: Storage ist gemeinsam, Compute ist bewusst isoliert, Governance ist von Anfang an Teil des Modells, und jede Komfortfunktion wird danach bewertet, welches konkrete Verhalten sie unter Last, Wachstum, Incident oder Audit verbessert. Wenn Teams Snowflake so lesen, verschiebt sich die Qualitaetsdiskussion weg von "funktioniert die Query?" hin zu "ist das Ergebnis für Betrieb, Kosten und Verantwortlichkeit tragfähig?" Genau das ist die Grundlage für produktionsreife Arbeit mit Snowflake.

---

1. Einordnung: Wann Snowflake die richtige Wahl ist

1.1. Problemklassen, in denen Snowflake typischerweise stark ist

Snowflake ist besonders sinnvoll, wenn mehrere Bedingungen gleichzeitig erfüllt sind:

- Analytische SQL-Workloads, die von elastischer Compute-Skalierung profitieren.
- Mehrere Teams oder Tools greifen auf denselben Datenbestand zu.
- Workload-Isolierung ist wichtig, damit BI, ELT, Data Science und Sharing sich nicht direkt stoeren.
- Semistrukturierte Daten (`VARIANT`, JSON, Avro, Parquet) sollen ohne separate Spezialplattform integriert werden.

- Datenprodukte sollen intern oder extern geteilt werden, ohne staendig neue Kopien anzulegen.
- Governance, Masking, Row Access und Klassifizierung sollen in derselben Plattform leben.

Typische Domaenen:

- Enterprise Analytics und BI.
- ELT/Medallion- oder Hub-and-Spoke-Architekturen.
- Zentrale Datenplattformen für mehrere Fachbereiche.
- Partner- und Kundensharing.
- Operative Analytik mit Minuten-Freshness.

1.2. Wann ein klassisches OLTP-System oft besser passt

Ein relationales OLTP-System passt meist besser, wenn du primär brauchst:

- Sehr niedrige, stabile Einzeltransaktionslatenz für viele Punkt-Updates.
- Fein abgestimmte Sperr- und Transaktionsmuster für operative Kernprozesse.
- Hohe Frequenz von kleinen `INSERT` / `UPDATE` -Operationen als Hauptlast.
- Applikationslogik, die eng mit Zeilenlocks und unmittelbarer Konsistenz arbeitet. Snowflake kann transaktionale Teilfaelle abdecken, insbesondere mit **Hybrid Tables**, ist aber im Kern keine generische Ersatzplattform für jedes klassische OLTP-Szenario.

1.3. Snowflake vs. Spark-/Lakehouse-first-Ansatz

Snowflake-Vorteile:

- Weniger Infrastruktur- und Cluster-Betrieb.
- Starke SQL-/Governance-/Sharing-Integration.
- Saubere Compute-Isolierung über Warehouses.
- Plattformdienste für Caching, Optimierung, Security und Datenfreigabe. **Lakehouse-/Spark-first-Vorteile (je nach Stack):**
- Mehr Offenheit bei Dateiformaten und eigener Engine-Kontrolle.
- Teilweise tiefere Kontrolle über Storage-Pfade, Job-Scheduling und Engine-Parameter.
- In manchen Teams bessere Passung, wenn bereits starkes Spark-/Databricks-Operating existiert.

Entscheidend ist nicht "Cloud Warehouse vs. Lakehouse", sondern:

- Wie stark du Plattformbetrieb selbst tragen willst.
- Wie wichtig dir integrierte Governance und Sharing-Mechaniken sind.
- Wie viele Nutzergruppen dasselbe Datenfundament brauchen.

1.4. Harte No-Go-Faelle für Snowflake

- Das Hauptsystem ist ein hochgradig transaktionales Kernsystem mit harter Millisekunden-Latenz als Primaranforderung.
- Das Team erwartet unbegrenzte Performance "einfach durch größeres Warehouse", ohne Datenmodell- und Query-Disziplin.
- Es gibt keine Bereitschaft, Kostenpfade aktiv zu steuern.
- Serverless Nebenkosten dürfen organisatorisch nicht existieren, können aber auch nicht beobachtet werden.
- Datenprodukt-Verantwortung ist ungeklaert und Sharing soll "mal schnell" aktiviert werden.

> **Anti-Pattern:** "Wir laden erstmal alles nach Snowflake und optimieren später." Ergebnis in der Praxis: wachsende Credit-Rechnungen, konkurrierende Workloads, unklare Ownership und späte Governance-Nacharbeit.

1.5. Entscheidungsmatrix (kompakt)

Kriterium	Snowflake passt	Snowflake passt eher nicht
Hauptlast	Analytik, ELT, Sharing, semistrukturierte Daten	Stark transaktionaler Kern-OLTP
Nutzergruppen	Viele Teams / Tools auf gleichem Datenfundament	Ein einzelner, eng begrenzter App-Use-Case
Governance	Rollen, Masking, Row Access, Klassifizierung wichtig	Governance ist extern oder kaum relevant
Betriebsmodell	Möglichst wenig eigener Clusterbetrieb	Vollkontrolle über Engine/Infra hat Prioritaet
Kostensteuerung	Aktives Warehouse- und Feature-Management möglich	Kosten dürfen praktisch nie schwanken

1.6. So setzt du es um - Checkliste

- Definiere vor der Plattformwahl mindestens drei echte Workload-Klassen.
- Dokumentiere, welche davon SQL-first und welche code-/engine-lastig sind.
- Lege fest, ob Sharing ein Kernziel oder nur optionales Nice-to-have ist.
- Bewerte explizit, ob Kosten pro Workload sichtbar und steuerbar sein müssen.
- Benenne einen Owner für Plattformkosten, nicht nur für Datenmodellierung.

1.7. Praxisvergleich: Snowflake vs. "wir strukturieren später"

Der häufigste Einfuehrungsfehler ist nicht ein falscher Parameter, sondern ein falscher Projektmodus. Viele Teams starten mit dieser impliziten Reihenfolge:

- Rohdaten zuerst laden.
- SQL-Modelle später strukturieren.
- Warehouses bei Bedarf größer machen.
- Governance nach dem ersten Audit ergänzen. Bei Snowflake führt diese Reihenfolge oft zu technischen und organisatorischen Schulden:
- Raw-, Core- und Serving-Layer verschwimmen.
- BI und ETL teilen sich versehentlich dieselbe Compute-Zone.
- Masking und Row Access müssen in bestehende Abhängigkeiten "hineingepresst" werden.
- Materialized Views, Dynamic Tables und Clustering werden aus Symptomdruck statt aus Architektur entschieden.

Die bessere Reihenfolge ist:

1. Datenprodukte und Workloads definieren.
2. Freshness, Sicherheitsniveau und Kostenbudgets je Datenprodukt festlegen.
3. Erst dann Tabellen-, Warehouse- und Pipeline-Modell bauen.
4. Vor Produktion Last- und Kostenprofile mit echten Nutzungspfaden prüfen.

1.8. Architekturentscheidungen sauber dokumentieren

Wenn Snowflake langfristig stabil und wirtschaftlich laufen soll, braucht jede Kernentscheidung eine kurze, pruefbare Begründung. Bewaehrte ADR-Punkte (Architecture Decision Record):

- Warum Snowflake und nicht Alternative X?
- Welche Workloads bekommen eigene Warehouses?
- Welche Datenprodukte sind canonical, welche nur abgeleitet?
- Wo werden Streams/Tasks verwendet, wo Dynamic Tables?
- Welche Daten dürfen geteilt, maskiert oder regional eingeschränkt werden?

Ergaenze jede ADR um Betriebsfolgen:

- Wie messen wir Kosten pro Workload?
- Wie erkennen wir Queueing, schlechtes Pruning oder Refresh-Probleme?
- Wer darf neue serverless Features aktivieren?
- Was ist das Rollback-Muster bei fehlerhaftem Schema oder Datenprodukt?

---

2. Architektur & Kernkonzepte

2.1. Die drei Schichten: Storage, Compute, Cloud Services

Snowflake organisiert sich in drei zentrale Ebenen:

- **Database Storage** für persistente Datenhaltung.
- **Compute** über virtuelle Warehouses.
- **Cloud Services** für Koordination, Sicherheit, Metadaten, Query-Optimierung und Verwaltung. Das ist mehr als eine Marketing-Grafik. Für die Praxis bedeutet es:
- Daten sind vom Compute entkoppelt.
- Compute kann pro Workload getrennt skaliert werden.
- Viele Verwaltungs- und Orchestrierungsaufgaben passieren ausserhalb deiner Warehouses.

2.2. Virtual Warehouses: Isolierung ist das eigentliche Superfeature

Ein Virtual Warehouse ist ein eigenständiger Compute-Cluster. Snowflake beschreibt explizit, dass Warehouses keine Compute-Ressourcen miteinander teilen und deshalb die Performance anderer Warehouses nicht direkt beeinflussen. Das ist in der Praxis entscheidend für:

- Trennung von ETL und BI.
- Isolierung von Ad-hoc-Exploration gegen produktive Dashboards.
- Eigene Kostenbudgets je Team oder Datenprodukt.

Wichtige Stellhebel:

- Warehouse-Groesse.
- `AUTO_SUSPEND` / `AUTO_RESUME`.
- Single-Cluster vs. Multi-Cluster.
- Query Acceleration.
- Dedizierte Warehouses für spezielle Lastprofile.

2.3. Micro-Partitions: Warum Snowflake nicht "einfach nur mehr CPU" ist

Alle Snowflake-Tabellen werden automatisch in Micro-Partitions zerlegt. Diese enthalten laut offizieller Dokumentation zwischen **50 MB und 500 MB unkomprimierte Daten** und werden spaltenorientiert gespeichert. Snowflake speichert Metadaten wie Wertebereiche und Distinct-Informationen pro Micro-Partition. Die Konsequenz:

- Gute Filter und gute Datenanordnung führen zu starkem Pruning.
- Schlechtes Pruning führt zu unnötigen Scans und damit zu mehr Laufzeit und Credits.
- Die wichtigste Performancefrage ist oft nicht "wie gross ist mein Warehouse?", sondern "wie viel muss wirklich gescannt werden?"

2.4. Pruning, natuerliche Ordnung und Clustering

Snowflake legt Daten automatisch in Micro-Partitions ab und sammelt dabei Clustering-Metadaten. Viele Tabellen funktionieren ohne explizite Clustering Keys bereits gut, wenn:

- Daten in einer für Queries guenstigen natuerlichen Reihenfolge geladen werden.
- Filter auf Spalten mit gutem Wertebereich/Selektivitaetsverhalten liegen.

Explizite Clustering Keys werden interessant, wenn:

- Sehr grosse Tabellen unter gezielten Filterpfaden stehen.
- Die natuerliche Ordnung durch viel DML oder inkrementelle Loads "verwischt" ist.
- Query-Latenzen trotz sinnvoller Warehouse-Groesse über Zeit erodieren.

Wichtig:

- Clustering verbessert nicht alles.
- Clustering erzeugt serverless Wartungskosten.
- Clustering sollte immer anhand echter Query-Pfade entschieden werden.

2.5. Cache-Ebenen: Result Cache vs. Warehouse Cache

Snowflake hat mehrere Wiederverwendungsmechanismen, die im Alltag oft verwechselt werden:

- **Persisted Query Results**: Ergebnisse können bei identischer Query und unveränderten Daten wiederverwendet werden; der Cache läuft standardmaessig nach **24 Stunden** ab.
- **Warehouse Cache**: Daten im lokalen Warehouse-Kontext können wiederverwendet werden, solange das Warehouse aktiv bleibt.

Praktische Folgen:

- Ein suspendiertes Warehouse verliert seinen Cache.
- Schon kleine Query-Aenderungen können Result-Reuse verhindern.
- Schnelle Dashboards profitieren oft mehr von stabilen Query-Texten und passendem `AUTO_SUSPEND` als von blindem Upsizing.

2.6. Tabellentypen bewusst wählen

Snowflake bietet unterschiedliche Tabellenarten mit unterschiedlichen Betriebsfolgen:

- **Permanent Tables**: Standard für produktive Kernobjekte; Time Travel plus Fail-safe.
- **Transient Tables**: Geringere Aufbewahrungs-/Resilienzeigenschaften, sinnvoll für bestimmte Staging- oder Zwischenschichten.

- **Temporary Tables**: Session-lokal bzw. kurzlebig.
- **External Tables / Iceberg-Szenarien**: Wenn Daten ausserhalb klassischer Snowflake-Storage-Muster liegen.
- **Hybrid Tables**: Für transaktionalere, niedrige Latenz, indexbasierte Random Reads/Writes, Row Locking sowie Unique-/Referential-Constraints. Die Wahl ist keine Formalitaet. Sie beeinflusst:
- Kosten.
- Wiederherstellbarkeit.
- Governance.
- Passung zum Workload.

2.7. Time Travel, Fail-safe und Zero-Copy-Klonen

Snowflake kombiniert mehrere Resilienz- und Produktivitaetsmechanismen:

- **Time Travel** für historische Abfragen, Restore und Clone.
- **Fail-safe** für durch Snowflake verwaltete Notfallwiederherstellung.
- **Zero-Copy Cloning** für schnelle Ableger von Datenbanken, Schemas oder Tabellen.

Wichtige Fakten:

- Standard-Time-Travel ist für alle Accounts vorhanden und beträgt standardmaessig **1 Tag**.
- Höhere Retention von mehr als 1 Tag bis **90 Tage** ist Enterprise Edition oder höher.
- Geklonte Objekte teilen sich zunächst denselben zugrunde liegenden Storage.

Das macht Klonen extrem wertvoll für:

- Dev/Test gegen produktionsnahe Daten.
- Reproduzierbare Release-Checks.
- Sichere Refactorings und Modellmigrationen.

2.8. Secure Data Sharing und Snowgrid

Snowflake kann Daten zwischen Accounts teilen, ohne die eigentlichen Daten zu kopieren oder zu verschieben. Der Consumer zahlt dabei nicht für Storage der geteilten Daten, sondern für sein eigenes Query-Compute. Mit **Snowgrid** erweitert Snowflake dieses Modell über Regionen und Clouds hinweg. Praktische Wirkung:

- Datenprodukte können als kontrollierte Freigaben modelliert werden.
- Konsumenten müssen nicht zwangslaufig eine Datei-, ETL- oder Exportkopie erhalten.
- Sharing wird zu einer Architekturentscheidung, nicht zu einem manuellen Exportprozess.

2.9. Mini-Projekt: Architekturentscheidung

Für die SaaS-Analytics-Plattform aus Abschnitt 0:

- `ingest_wh` für Batch und Snowpipe-nahe Ladepfade.
- `transform_wh` für Core-/Serving-Transformationen.
- `bi_wh` für Dashboards.
- `sandbox_wh` für explorative Data-Science-Abfragen.
- Permanente Tabellen für kuratierte Kernschichten.
- Transiente Objekte für bestimmte Landing-/Arbeitszonen.
- Secure Share für Partnerzugriff auf freigegebene Marts.

2.10. So setzt du es um - Checkliste

- Dokumentiere pro Warehouse Zweck, Owner, `AUTO_SUSPEND`, Größe und Kostenregel.
- Prüfe für grosse Tabellen systematisch, ob Pruning gut genug ist, bevor du größer skalierst.
- Entscheide Tabellenarten bewusst statt beim Default zu bleiben.
- Nutze Klone für Tests und risikoarme Migrationen.
- Definiere, welche Datenprodukte kopiert werden dürfen und welche nur geteilt werden.

2.11. Architektur-Risiken früh erkennen

Architekturbedingte Probleme in Snowflake sind oft schleichend:

- Immer größere Warehouses ohne entsprechende Query-Verbesserung.
- BI und ETL beeinflussen sich indirekt über gemeinsam genutztes Warehouse.
- Clustering wird breit eingeschaltet, ohne Kosten-Nutzen-Pruefung.
- Materialized Views oder Dynamic Tables werden angelegt, aber nie wieder wirtschaftlich hinterfragt.

Frühe Signale:

- Stark steigende `bytes_scanned` bei gleichbleibender fachlicher Ergebnismenge.
- Sichtbare Warehouse-Queues.
- Wachsende serverless Verbraeuche ohne klares Ownership-Modell.
- Freshness-Probleme bei Datenprodukten trotz "automatischer" Aktualisierung.

2.12. Architektur-Check vor Produktion

Vor dem ersten echten Produktionsverkehr sollte ein technischer Architektur-Check zwingend durchlaufen werden. Checkpunkte:

- Workload-Isolierung über Warehouses ist dokumentiert.
- Time-Travel-/Retention-Politik ist je Datenklasse definiert.
- Rolle, Datenprodukt und Freigabemodell sind sauber getrennt.
- Batch-, CDC- und Freshness-Pfade sind technisch und organisatorisch geklärt.
- Kostenverantwortung für Warehouses und serverless Features ist benannt.
- Query- und Incident-Monitoring ist aktiv.

2.13. Architektur-Entscheidungen an SLO und Kosten koppeln

Architektur wird belastbar, wenn sie direkt an messbare Ziele gekoppelt ist. Beispiel:

- SLO: Dashboard-Freshness < 5 Minuten.
- Architekturfolge: Dynamic Table oder stream-getriebene Task-Kette statt manueller Batch.
- Betriebsfolge: eigenes Transform-Warehouse plus Refresh-Monitoring.
- Kostenfolge: klares Budget für Refresh-Compute und serverless Nebendienste. Wenn diese Ebenen nicht zusammen gelesen werden, werden Architekturentscheidungen zu Meinungen.

---

3. Datenmodellierung in der Praxis

3.1. Modelliere von Datenprodukten und Zugriffspfaden rueckwaerts

In Snowflake ist das Datenmodell oft weniger durch Normalform-Dogmen als durch Datenproduktvertraege bestimmt. Die zentrale Frage lautet nicht nur: "Welche Tabellen habe ich?" Sondern: "Welche Schichten, Zugriffspfade, Freshness-Ziele und Sicherheitsregeln sollen stabil betrieben werden?" Deshalb lohnt es sich, für jede Haupttabelle zu dokumentieren:

- Wer nutzt sie?
- Wie oft ändert sie sich?
- Welche typischen Filter, Joins und Aggregationen existieren?
- Muss sie geteilt, maskiert oder regional eingeschränkt werden?

3.2. Raw, Core, Serving: Schichten statt Daten-Sumpf

Ein bewaehrtes Muster:

- **Raw/Landing**: möglichst nah an der Quelle.
- **Core/Conformed**: bereinigt, typisiert, fachlich konsolidiert.
- **Serving/Marts**: konsumorientiert für BI, APIs, Partner oder Modelle.

Vorteile:

- Ownership wird klarer.
- Fehler lassen sich schneller eingrenzen.
- Governance lässt sich je Schicht anders umsetzen.
- Performance-Tuning kann auf Serving-Objekte fokussiert werden.

3.3. Breite Tabellen sind nicht automatisch schlecht

Snowflake ist analytisch gepraegt. Deshalb sind breitere, konsumorientierte Tabellen oft sinnvoller als uebernormalisierte Designs, wenn dadurch:

- weniger komplexe Joins nötig sind,
- Zugriffspfade stabiler werden,
- BI- und Self-Service-Nutzung einfacher wird.

Wichtig ist jedoch:

- Nicht jede Redundanz ist kostenlos.
- Aenderungslogik, Refresh-Aufwand und Governance müssen mitgedacht werden.
- Eine breite Serving-Tabelle braucht eine belastbare Core-Quelle.

3.4. Semistrukturierte Daten mit `VARIANT` bewusst einsetzen

Snowflake kann semistrukturierte Daten gut verarbeiten. Das ist hilfreich für:

- Event-Payloads.
- API-Rohdaten.
- Attribute, die sich häufig ändern.

Aber:

- `VARIANT` ist kein Freifahrtschein für dauerhaft ungepflegte Rohmodelle.
- Spätestens für stabile Datenprodukte sollten häufig genutzte Felder typisiert, benannt und fachlich verankert werden.
- Gute Governance und gute BI-Nutzung verlangen auf Dauer explizite Semantik.

3.5. Clustering Keys nur für echte Problemstellen

Snowflake empfiehlt Clustering nicht pauschal für jede Tabelle. Sinnvoll wird ein Clustering Key typischerweise, wenn:

- Tabellen sehr gross sind,
- selektive Filter auf wenigen Spalten häufig vorkommen,
- natuerliche Ordnung für Queries nicht mehr ausreicht.

Unsinnige Muster:

- Clustering auf kleine oder selten genutzte Tabellen.
- Clustering, bevor Query-Profile und Pruning gemessen wurden.
- Zu viele Clustering-Dimensionen ohne klares Abfrageziel.

Merksatz:

- Erst Query-Verhalten messen.
- Dann prüfen, ob Datenmodell, Filter und Ladeordnung reichen.
- Clustering erst danach gezielt einsetzen.

3.6. Permanente, transiente und temporäre Schichten nicht vermischen

Ein häufiger Modellierungsfehler ist, Tabellenarten nur technisch zu betrachten. In Wirklichkeit bilden sie ein Betriebsversprechen ab:

- Permanent: hohe Wiederherstellbarkeit und Nachvollziehbarkeit.
- Transient: geringerer Schutz, oft passend für bestimmte Zwischenstufen.
- Temporary: nur für kurzlebige, explizit temporäre Arbeitskontexte. Wenn diese Semantik fehlt, entstehen später Konflikte zwischen Finance, Operations und Security.

3.7. Hybrid Tables für gezielte operative Teilfaelle

Hybrid Tables sind für niedrige Latenz, hohen Durchsatz, indexbasierte Random Reads/Writes, Row Locking und Unique-/Referential-Integrity optimiert. Sie sind interessant, wenn:

- eine Plattform bewusst transaktionale und analytische Daten näher zusammenfuehren will,
- ein Teil-Use-Case operativer ist als klassisches DWH,
- analytische Auswertung auf derselben Plattform bleiben soll.

Aber:

- Hybrid Tables sind kein Grund, Snowflake zum universellen OLTP-Ersatz zu erklären.
- Nicht alle Features von Standardtabellen stehen dort identisch zur Verfügung.

3.8. Views, Materialized Views und Dynamic Tables sauber unterscheiden

Diese drei Konstrukte lösen unterschiedliche Probleme:

- **View**: logische Abstraktion ohne Speicherung der Ergebnisse.
- **Materialized View**: vorgerechnete Teilmenge/Projektion für häufige, teure Pfade.
- **Dynamic Table**: deklaratives Refresh-Modell für Datenpipelines und materielle Zieltabellen.

Faustregeln:

- View für Semantik und Zugriffskapselung.
- Materialized View für oft genutzte, teure, relativ stabile Query-Muster.
- Dynamic Table für regelmäßig aktualisierte Pipeline-Ergebnisse mit Ziel-Freshness.

3.9. Mini-Projekt: Schichtenmodell

Für das SaaS-Szenario:

- `raw.events_app`, `raw.billing_files`, `raw.support_tickets`.
- `core.customers`, `core.subscriptions`, `core.invoice_lines`, `core.ticket_facts`.
- `serving.mrr_daily`, `serving.support_sla_breaches`, `serving.product_usage_5m`.
- Partnerfreigabe nur aus `share.partner_usage_mart`.

So bleiben:

- Quellnahe Last,
- fachliche Konsolidierung,
- konsumnahe Modelle sauber getrennt.

3.10. So setzt du es um - Checkliste

- Lege für jede Tabelle eine Schichtzuordnung fest.
- Dokumentiere häufige Filter, Join-Pfade und Freshness-Anforderungen.
- Typisiere häufig genutzte Felder aus `VARIANT` explizit aus.
- Setze Clustering nur nach Messung von Pruning- und Laufzeitproblemen ein.
- Trenne permanente Kernobjekte von kurzlebigen Arbeitsobjekten.

3.11. Haefige Anti-Pattern in Snowflake-Datenmodellen

- Raw-Tabellen werden direkt als BI-Quellen verwendet.
- Alle Attributaenderungen bleiben dauerhaft nur in JSON-Feldern verborgen.
- Jede langsame Tabelle bekommt reflexartig ein größeres Warehouse statt besseres Modell.
- Clustering Keys werden broad-spectrum auf viele Tabellen ausgerollt.
- Materialized Views entstehen, ohne dass Nutzungshaeufigkeit und Wartungskosten geprüft wurden.
- Partner-Sharing greift auf interne Core-Tabellen statt auf explizite Datenprodukte zu.

3.12. Datenprodukt-Vertraege explizit machen

Ein produktionsreifes Snowflake-Modell beschreibt nicht nur DDL, sondern einen Vertrag:

- Fachlicher Zweck.
- Erwartete Aktualisierungsfrequenz.
- Erlaubte Nutzergruppen.
- Sensible Spalten und Schutzmechanismen.

- Downstream-Abhaengigkeiten. Das macht Refactorings, Audits und Incident-Kommunikation deutlich einfacher.

---

4. SQL, Laden & Transformieren

4.1. Basisobjekte für saubere Ladepfade

Viele Snowflake-Projekte werden unuebersichtlich, weil Stages, File Formats und Ladeobjekte zu spät strukturiert werden. Bewaehrt hat sich ein klares Basisset aus:

- Datenbank und Schema je Zone.
- Benannte File Formats.
- Benannte Stages.
- Konsistente Naming-Konventionen für Ladeobjekte.

Beispiel:

```
CREATE OR REPLACE WAREHOUSE ingest_wh
  WITH WAREHOUSE_SIZE = 'MEDIUM'
       AUTO_SUSPEND = 60
       AUTO_RESUME = TRUE;

CREATE OR REPLACE DATABASE analytics;
CREATE OR REPLACE SCHEMA analytics.raw;

CREATE OR REPLACE FILE FORMAT analytics.raw.json_ff
  TYPE = JSON
  STRIP_OUTER_ARRAY = TRUE;

CREATE OR REPLACE STAGE analytics.raw.app_stage
  URL = 's3://company-prod-events/app/'
  STORAGE_INTEGRATION = aws_prod_int
  FILE_FORMAT = analytics.raw.json_ff;
```

4.2. `COPY INTO` für kontrollierte Batch-Loads

`COPY INTO <table>` ist der klassische, robuste Batch-Load-Pfad. Stark ist er, wenn:

- Dateien geordnet in Stages ankommen,
- Ladefenster klar sind,
- Wiederholbarkeit und Nachvollziehbarkeit wichtiger sind als Sekunden-Latenz.

Beispiel:

```
COPY INTO analytics.raw.app_events_raw
FROM @analytics.raw.app_stage
PATTERN = '.*\\.json\\.gz'
ON_ERROR = 'CONTINUE';
```

Praxisregeln:

- Ladefehler und Reprocessing-Pfade vorher definieren.
- File Formats nicht inline-chaotisch verteilen.
- Landing-Tabellen von Core-Transformationen trennen.

4.3. Snowpipe für kontinuierliche Micro-Batches

Snowpipe laedt Dateien, sobald sie in einer Stage verfügbar sind. Offizielle Doku beschreibt Snowpipe als serverless und auf kontinuierliche Micro-Batches ausgelegt. Das ist sinnvoll, wenn:

- Dateien häufig eintreffen,
- minutennahe Verfügbarkeit gebraucht wird,
- du nicht staendig eigene Polling-Jobs bauen willst.

Merke:

- Snowpipe ist kein Allheilmittel für jeden Ingest.
- Governance, Schemaentwicklung und Nachverarbeitung bleiben trotzdem deine Aufgabe.

4.4. Snowpipe Streaming für echte niedrigere Latenz

Wenn Daten nicht erst als Dateien landen sollen, sondern row-level kontinuierlich einfliessen, ist Snowpipe Streaming interessant. Es passt typischerweise zu:

- Eventing.
- Telemetrie.
- kontinuierlichen App- oder Plattformdaten.

Wichtig ist die Entscheidung entlang von:

- Datenvolumen,
- Latenzanspruch,
- Quellsystemfaehigkeit,
- Betriebskomplexitaet.

4.5. Streams: CDC als Bookmark-Modell

Ein Stream speichert nicht die Tabelle selbst, sondern einen Offset und nutzt Change-Tracking-Metadaten, um DML-Aenderungen sichtbar zu machen. Das ist das passende Modell für:

- inkrementelle ELT-Pfade,
- Upsert-Merges,
- eventgetriebene Verarbeitung.

Beispiel:

```
CREATE OR REPLACE STREAM analytics.raw.orders_stream
  ON TABLE analytics.raw.orders_raw;
```

Praxisrelevant:

- Streams sind ein CDC-Werkzeug, kein Archiv.
- Sie müssen in ein klares Konsummuster eingebettet werden.
- Nicht konsumierte oder organisatorisch vergessene Streams werden schnell zum Betriebsproblem.

4.6. Tasks: geplant, getriggert, serverless oder user-managed

Tasks können:

- nach festem Zeitplan laufen,

- auf neue Stream-Daten reagieren,
- serverless oder auf einem benannten Warehouse ausgeführt werden.

Snowflake dokumentiert beides klar:

- Fixe Schedules.
- Triggered Tasks, z. B. mit `SYSTEM$STREAM_HAS_DATA(...)`.

Beispiel:

```
CREATE OR REPLACE TASK analytics.transform.merge_orders
  WAREHOUSE = transform_wh
  SCHEDULE = '5 MINUTES'
  WHEN SYSTEM$STREAM_HAS_DATA('analytics.raw.orders_stream')
AS
MERGE INTO analytics.core.orders t
USING (
  SELECT
    order_id,
    customer_id,
    status,
    total_amount,
    updated_at
  FROM analytics.raw.orders_stream
) s
ON t.order_id = s.order_id
WHEN MATCHED THEN UPDATE SET
  customer_id = s.customer_id,
  status = s.status,
  total_amount = s.total_amount,
  updated_at = s.updated_at
WHEN NOT MATCHED THEN INSERT (
  order_id, customer_id, status, total_amount, updated_at
) VALUES (
  s.order_id, s.customer_id, s.status, s.total_amount, s.updated_at
);
```

Entscheidungslogik:

- **User-managed Task**: wenn du das Warehouse bewusst kontrollieren willst.
- **Serverless Task**: wenn Betriebsaufwand klein bleiben soll und Lastprofile dazu passen.

4.7. Dynamic Tables: deklarative Pipeline-Ergebnisse

Dynamic Tables aktualisieren sich automatisch auf Basis einer Query und eines Freshness-Ziels (`TARGET_LAG`). Offizielle Doku beschreibt sie als Vereinfachung für Transformation und Pipeline-Management ohne manuelles Scheduling. Beispiel:

```
CREATE OR REPLACE DYNAMIC TABLE analytics.serving.mrr_daily
  TARGET_LAG = '5 MINUTES'
  WAREHOUSE = transform_wh
AS
SELECT
  invoice_date,
  customer_region,
  SUM(mrr_amount) AS mrr
FROM analytics.core.subscription_daily
GROUP BY invoice_date, customer_region;
```

Dynamic Tables sind besonders sinnvoll, wenn:

- ein deklaratives Ziel wichtiger ist als feingranulare Job-Orchestrierung,

- mehrere Pipeline-Stufen logisch verkettet werden sollen,
- Freshness als Zielwert ausdrueckbar ist.

Sie sind weniger ideal, wenn:

- du sehr spezielle Kontrolllogik brauchst,
- Prozessgrenzen extrem explizit sein müssen,
- Fehlerbehandlung stark individuell ist.

4.8. Materialized Views für wiederkehrende teure Muster

Materialized Views speichern vorgerechnete Ergebnisse einer Query. Sie sind besonders sinnvoll, wenn:

- dieselben teuren Muster oft laufen,
- die Ergebnismenge relativ klein zur Basistabelle ist,
- sich die Basistabelle nicht extrem häufig ändert.

Wichtig:

- Materialized Views bringen Wartungs- und ggf. Storage-Kosten mit.
- Sie sind ein Performance-Werkzeug, kein Ersatz für schlechte Grundmodellierung.

4.9. Snowpark, dbt und SQL-Aufteilung

Snowflake ist stark in SQL, aber nicht nur dort. In der Praxis hat sich oft folgende Arbeitsteilung bewaehrt:

- **SQL** für klare, nachvollziehbare Transformationen.
- **dbt** für modulare SQL-Transformationen, Tests und Deployments.
- **Snowpark** für komplexere Logik, Data-Science-nahe Transformationen oder Sprachintegration.

Merksatz:

- Nicht jede Logik gehört in Python, nur weil sie technisch dort möglich ist.
- Nicht jede Pipeline gehört in pures SQL, nur weil es bequem erscheint.

4.10. Musterwahl: Batch, Micro-Batch, CDC oder deklarative Freshness?

Eine praxistaugliche Auswahlhilfe:

- `COPY INTO`: wenn feste Ladefenster und Batch-Nachvollziehbarkeit im Vordergrund stehen.
- Snowpipe: wenn dateibasiert, aber kontinuierlicher.
- Streams + Tasks: wenn Änderungen explizit verarbeitet und orchestriert werden sollen.
- Dynamic Tables: wenn deklarative Refresh-Ziele und weniger Scheduling-Code gefragt sind.
- Materialized Views: wenn Query-Beschleunigung, nicht Pipeline-Orchestrierung, das Ziel ist.

4.11. So setzt du es um - Checkliste

- Trenne Dateiformat, Stage und Load-Logik sauber.
- Nutze `COPY INTO` bewusst für nachvollziehbare Batch-Pfade.
- Setze Streams nur dort ein, wo ein echter CDC-Konsument existiert.
- Waehle zwischen Tasks und Dynamic Tables nach Steuerungsbedarf statt nach Trend.
- Miss bei jeder neuen Komfortfunktion die Kostenwirkung mit.

4.12. Migrationspfad: von taeglichem Batch zu Minuten-Freshness

Ein realistischer Ausbaupfad ist oft:

1. Start mit geordneten Batch-Loads über `COPY INTO`.
2. Danach Snowpipe für haeufigere Dateianlieferungen.
3. Inkrementelle Core-Verarbeitung per Streams/Tasks.
4. Serving-Ebene via Dynamic Tables oder gezielte Materialized Views. So bleibt die Plattform kontrollierbar, statt alles gleichzeitig zu "verstreamen".

---

5. Datenprodukte, Zusammenarbeit & Resilienz

5.1. Workloads als Produkte denken

Snowflake wird stabiler, wenn du nicht nur Tabellen, sondern Datenprodukte modellierst. Ein Datenprodukt umfasst mindestens:

- definierte Konsumenten,
- garantierte Freshness,
- klares Sicherheitsprofil,
- benannte Eigentuemerschaft,
- messbare Betriebs- und Kostenfolgen.

Das ist wichtig, weil Snowflake technische Leichtigkeit erzeugt:

- schnell ein Share,
- schnell ein Clone,
- schnell ein neues Warehouse,
- schnell eine Dynamic Table. Ohne Produktdenken wachsen daraus schnell unkoordinierte Parallelwelten.

5.2. Secure Data Sharing statt Export-Orgie

Secure Data Sharing funktioniert ohne Kopie oder Transfer der eigentlichen Daten zwischen Accounts. Das ist architektonisch stark, weil:

- keine Exportdateien verwaltet werden müssen,
- keine redundanten Speicherpfade beim Consumer entstehen,
- Governance näher am Original-Datenprodukt bleibt.

Aber:

- Ein Share ist kein Ersatz für fachliche Schnittstellenbeschreibung.
- Geteilte Objekte brauchen dieselbe Sorgfalt wie externe APIs.

5.3. Reader Accounts und kontrollierte Freigabe

Wenn ein Empfänger keinen Snowflake-Account hat, können Reader-Account-Szenarien sinnvoll sein. Das ist praktisch für:

- Pilotpartner,

- begrenzte Datendienste,
- kontrollierte externe Nutzung.

Entscheidend bleibt:

- Wer bezahlt Compute?
- Wer verwaltet Rollen und Freigaben?
- Wie werden PII, Regionseinschraenkungen und Supportfaelle abgebildet?

5.4. Clones für Dev/Test, QA und Release-Sicherheit

Zero-Copy Cloning ist eines der produktivsten Snowflake-Features. Sinnvolle Einsatzfaelle:

- Release-Proben gegen produktionsnahe Daten.
- Vergleich alter/neuer Modelle.
- schnelle Isolierung eines Datenzustands für Incident-Analyse.
- QA-Umgebungen mit realistischen Datenvolumina.

Aber:

- Klone sind kein kostenloser Ersatz für Datenlebenszyklusmanagement.
- Grants, Clustering-Status und Nebeneffekte müssen bewusst mitgelesen werden.

5.5. Replication, Failover und Snowgrid

Snowgrid adressiert cross-region- und cross-cloud-Szenarien. In der Praxis relevant für:

- Business Continuity.
- regionale Datenprodukte.
- kontrollierte Distributionsmodelle über Clouds hinweg.

Wichtige Leitfrage:

- Welcher Teil deiner Plattform braucht wirklich Resilienz über Region oder Cloud hinaus? Nicht jedes Objekt braucht denselben Resilienzgrad.

5.6. Mini-Projekt: Produkt- und Freigabemodell

Für das SaaS-Szenario:

- `serving.product_usage_5m` für interne operative Dashboards.
- `serving.finance_daily_close` für Finance mit strengem Abschlussprozess.
- `partner.partner_usage_monthly` als explizites Share-Produkt.
- Eigener Clone von `core` für QA vor jedem größeren Modell-Release.

Damit wird klar:

- interne Nutzung,
- externe Nutzung,
- Governance,
- Release-Sicherheit laufen über unterschiedliche Verträge.

5.7. So setzt du es um - Checkliste

- Beschreibe jedes externe Share als Datenprodukt mit fachlichem Vertrag.

- Nutze Klone bewusst für Release-, QA- und Incident-Zwecke.
- Trenne interne Serving-Objekte von externen Share-Objekten.
- Definiere Resilienz nicht global, sondern pro kritischer Datenklasse.
- Dokumentiere Kosten- und Supportverantwortung für jeden Consumer-Pfad.

5.8. Zusammenarbeit ohne Plattformchaos

Snowflake erleichtert Zusammenarbeit technisch enorm. Gerade deshalb braucht es Spielregeln:

- Wer darf neue Warehouses anlegen?
- Wer darf Materialized Views, Clustering oder Search Optimization aktivieren?
- Wer darf Shares erstellen?
- Wer entscheidet über `permanent` vs. `transient`? Ohne diese Leitplanken wird Kollaboration schnell zu Kosten- und Governance-Drift.

5.9. Datenprodukte versionieren, nicht nur Tabellen

Tabellen- und View-Namen allein reichen für produktionsreife Zusammenarbeit selten aus. Versionierungsfragen:

- Wie wird ein Breaking Change angekuendigt?
- Gibt es Parallelbetrieb für ein Nachfolgemodell?
- Wie wird ein Share migriert?
- Welche Freshness-/Semantikgarantien bleiben stabil? Das ist besonders wichtig, wenn externe Partner oder mehrere interne Domainen auf denselben Produkten arbeiten.

---

6. Betrieb, Monitoring & Troubleshooting

6.1. Monitoring-Grundsatz

Snowflake ist zwar stark gemanagt, aber nicht "selbsterklaerend". Du brauchst trotzdem aktive Beobachtung auf mehreren Ebenen:

- Query-Ebene.
- Warehouse-Ebene.
- Pipeline-Ebene.
- Governance-/Security-Ebene.
- Kostenebene.

6.2. Kernquellen für Observability

Zentrale Beobachtungsorte sind u. a.:

- `QUERY_HISTORY`
- Query Profile in Snowsight
- Warehouse Activity / Queueing
- `TASK_HISTORY` bzw. `SERVERLESS_TASK_HISTORY`
- Ladehistorien für Batch und Snowpipe-nahe Pfade

- Budget- und Metrierungsansichten Snowflake hebt beim Query Profile ausdrücklich hervor, dass dort die teuersten Ausfuehrungsknoten sichtbar werden. Das ist oft der schnellste Weg, um echte Engpaesse statt Vermutungen zu sehen.

6.3. Queueing: Der Klassiker bei gemeinsam genutztem Compute

Wenn zu viele Queries gleichzeitig auf ein Warehouse treffen, werden spätere Queries gequeued. Snowflake dokumentiert dies klar als Folge erschoepfter Compute-Ressourcen. Praxisfolgen:

- Nutzer nehmen "Snowflake ist langsam" wahr.
- Tatsächlich ist oft nicht die Query selbst das Hauptproblem, sondern Konkurrenz auf dem Warehouse.

Gegenmassnahmen:

- Workloads trennen.
- Warehouse vergroessern, wenn wirklich Compute fehlt.
- Bei Bedarf Multi-Cluster einsetzen.
- Spiky Einzelqueries aus produktiven Pfaden herausziehen.

6.4. Cache-Fehlannahmen erkennen

Typische Missverständnisse:

- "Gestern war die Query schnell, also muss das Modell gut sein."
- "Ein größeres Warehouse bringt immer dieselbe Beschleunigung."
- "Der Cache wirkt auch nach Suspend weiter."

Tatsächlich:

- Suspend leert den Warehouse Cache.
- Result Reuse verlangt sehr enge Bedingungen.
- Wiederholte Schnelligkeit kann ein Cache-Artefakt statt echte Robustheit sein.

6.5. Pruning-Probleme systematisch lesen

Wenn `bytes_scanned` hoch bleiben oder Laufzeiten steigen:

- Sind Filter selektiv genug?
- Liegen Castausdruecke oder Funktionen auf den Filterspalten?
- Ist die Datenordnung für die Hauptabfragen noch guenstig?
- Braucht die Tabelle wirklich einen Clustering Key?
- Ist Search Optimization sinnvoller als pauschales Upsizing? Das ist der Snowflake-typische Diagnosepfad. Nicht zuerst CPU, sondern zuerst Scanlogik.

6.6. Pipeline-Troubleshooting: Streams, Tasks, Dynamic Tables

Typische Fehlerbilder:

- Stream wird nicht konsumiert oder fachlich vergessen.
- Task ist suspendiert, falsch getriggert oder läuft auf unpassendem Warehouse.
- Dynamic Table verfehlt `TARGET_LAG`, wird uebersprungen, langsam oder fehlerhaft.
- Ladepfade leiden unter Schema-Drift oder unklaren File Formats.

Praxisregel:

- Jedes Produktionsdatenprodukt braucht ein Runbook, das Source, Refresh-Modell, Owner, Monitoring-Signale und nächste Aktionen nennt.

6.7. Kosten-Incidents sind echte Betriebs-Incidents

Viele Teams behandeln Kosten getrennt vom Betrieb. In Snowflake ist das ein Fehler. Kosten-Incidents entstehen z. B. durch:

- falsch dimensionierte oder dauerlaufende Warehouses,
- starke Queueing-Reaktion mit hektischem Upsizing,
- aktivierte serverless Features ohne Kostenleitplanke,
- unbemerkte Wartungskosten von Clustering oder Materialized Views,
- experimentelle Ad-hoc-Last auf produktionsnahen Compute-Pfaden.

6.8. Runbook-Vorlage pro Datenprodukt

Eine einfache, aber wirksame Vorlage:

- Datenproduktname und Business-Zweck.
- Quellobjekte und Refresh-Modell.
- Warehouse bzw. serverless Abhängigkeiten.
- Normalbereich für Freshness, Laufzeit und Credit-Verbrauch.
- Sichtbare Fehlerbilder.
- Erste 5 Diagnoseorte.
- Eskalationskette.

6.9. So setzt du es um - Checkliste

- Beobachte Query-, Warehouse-, Pipeline- und Kostenebene getrennt.
- Nutze Query Profile für echte Engpassanalyse.
- Tracke Queueing explizit, nicht nur Query-Dauer.
- Baue pro Produktionsdatenprodukt ein kurzes Runbook.
- Behandle Kostenanomalien mit derselben Ernsthaftigkeit wie technische Ausfaelle.

6.10. Post-Incident-Lernen für Snowflake-Teams

Nach jedem Incident sollte es eine kurze technische Nachbereitung geben:

1. Was war der erste messbare Indikator?
2. War das Problem Scan/Pruning, Queueing, Refresh, Governance oder Kosten?
3. Welche Architekturannahme war falsch oder unvollständig?
4. Welche Guardrail oder welches Budget hätte früher geholfen?
5. Welche Änderung folgt konkret an Warehouse-, Datenmodell- oder Governance-Design? So entsteht aus jedem Incident ein stabileres Plattformmodell.

---

7. Performance, Skalierung & Kostenkontrolle

7.1. Warehouse-Sizing: so klein wie stabil, so gross wie nötig

Warehouse-Groesse ist ein wirksamer Hebel, aber nicht der erste und nicht der einzige. Pragmatisches Vorgehen:

1. Query- und Pruning-Verhalten verstehen.
2. Workload-Isolierung sicherstellen.
3. Erst dann Warehouse-Groesse feinjustieren. Ein zu kleines Warehouse führt zu Queueing und langen Laufzeiten. Ein zu grosses Warehouse verschwendet Credits, wenn das eigentliche Problem woanders liegt.

7.2. `AUTO_SUSPEND`: Performance- und Kostenhebel zugleich

Snowflake dokumentiert explizit, dass der Warehouse Cache beim Suspend verworfen wird. Daraus folgt:

- Für **Tasks** empfiehlt Snowflake unmittelbare Suspendierung.
- Für **Dev/DataOps/Data Science** nennt die Doku grob etwa 5 Minuten als sinnvollen Richtwert.
- Für **BI-/SELECT-Warehouses** empfiehlt die Doku oft mindestens etwa 10 Minuten, um Cache-Nutzen zu erhalten.

Wichtig:

- Diese Werte sind Heuristiken, keine Naturgesetze.
- Sie müssen mit realem Anfrageabstand und Kostenrahmen abgeglichen werden.

7.3. Multi-Cluster Warehouses

Wenn Queues durch parallele Lastspitzen entstehen, kann ein Multi-Cluster-Warehouse sinnvoll sein. Snowflake nennt dies ausdrücklich als Mittel gegen Queueing-Spitzen. Beachte:

- Multi-Cluster ist Enterprise-Feature.
- Es ist ein Parallelitaetswerkzeug, kein Ersatz für schlechte Queries.
- Scaling Policy beeinflusst die Kostencharakteristik.

7.4. Search Optimization Service

Search Optimization ist für bestimmte selektive Query-Muster gedacht, insbesondere Punkt- oder kleine Teilmengenabfragen auf grossen Tabellen. Das ist interessant für:

- hochselektive Dashboard-Filter,
- operative Lookup-Muster,
- datenproduktnahe Suchen auf grossen Tabellen. Es ist nicht pauschal sinnvoll für alle analytischen Abfragen.

7.5. Query Acceleration Service (QAS)

QAS kann Teile bestimmter Workloads beschleunigen, indem es outlier-lastige Scan-/Filteranteile auf gemeinsame Compute-Ressourcen auslagert. Laut Doku eignet es sich besonders für:

- Ad-hoc-Analytik,
- unvorhersehbare Datenvolumina pro Query,

- grosse Scans mit selektiven Filtern.

Wichtig:

- QAS kann Credit-Verbrauch erhöhen.
- Nicht jede Query ist geeignet.
- Eignung sollte über `QUERY_ACCELERATION_ELIGIBLE` bzw. `SYSTEM$ESTIMATE_QUERY_ACCELERATION` datenbasiert geprüft werden.

7.6. Automatic Clustering nur mit Kostenbewusstsein

Automatic Clustering verwaltet Reclustering serverless im Hintergrund. Das ist bequem, aber nicht kostenlos. Snowflake empfiehlt deshalb ausdrücklich, mit ein oder zwei ausgewählten Tabellen zu starten und die Wirkung zu beobachten. Praxisregeln:

- Nur für Tabellen mit echtem Performancebedarf.
- Vorher und nachher Query-Verhalten messen.
- Kosten baseline-basiert verfolgen.

7.7. Materialized Views und Wartungskosten

Materialized Views sind stark, wenn:

- dieselben teuren Muster oft laufen,
- die Ergebnismenge vergleichsweise klein ist,
- Basisdaten nicht extrem häufig ändern.

Sie sind schwach, wenn:

- das Muster selten ist,
- sich die Basistabelle sehr häufig ändert,
- eigentliche Probleme in Modell oder Warehouse-Konkurrenz liegen.

7.8. Budgets und Resource Monitors

Kostenkontrolle in Snowflake braucht mindestens zwei Denkarten:

- **Budgets** für breitere Credit-Ueberwachung.
- **Resource Monitors** für Warehouses.

Wichtige Unterscheidung:

- Das **Account Budget** überwacht Ausgaben für alle Credit-Verbraeuche im Account.
- **Resource Monitors** kontrollieren Warehouse-Verbrauch, aber **nicht** die Snowflake-provided Compute-Ressourcen für serverless Features wie Snowpipe, Automatic Clustering oder Materialized Views.

Konsequenz:

- Wer nur Resource Monitors setzt, sieht nur einen Teil des Problems.

7.9. Mini-Projekt: Tuning-Loop

1. `bi_wh` und `transform_wh` trennen.
2. Queueing und `bytes_scanned` beobachten.

3. Für Hot-Queries prüfen: besseres Modell, Materialized View, Search Optimization oder QAS?
4. `AUTO_SUSPEND` je Workload feinjustieren.
5. Account Budget plus Warehouse-Monitore aktivieren.

7.10. So setzt du es um - Checkliste

- Optimiere zuerst Pruning und Workload-Isolierung, dann Warehouse-Groesse.
- Setze `AUTO_SUSPEND` workload-spezifisch.
- Nutze Multi-Cluster nur gegen echte Parallelitaetsspitzen.
- Prüfe Search Optimization und QAS anhand realer Query-Muster.
- Kombiniere Budgets mit Resource Monitors statt nur eines von beiden.

7.11. Benchmarking, das wirklich Aussagen liefert

Ein belastbarer Benchmark bildet mindestens ab:

- echte Filter- und Join-Pfade,
- typische Parallelitaet,
- Cache-kalte und Cache-warmer Szenarien,
- Freshness-/Refresh-Last,
- Kosten pro Ausfuehrungspfad.

Wichtig sind nicht nur Mittelwerte, sondern:

- P95/P99 für wichtige Dashboards oder APIs,
- Queueing unter Peak,
- Credit-Verbrauch pro Workload,
- Stabilität nach Schema- oder Pipeline-Aenderungen.

7.12. Kostenklassifizierung für Snowflake

Ein praxisnahes Modell:

- **Tier A**: kritische BI-/Finance-Workloads, klare SLOs, dedizierte Warehouses.
- **Tier B**: regulare Transformationspfade, feste Budgets und klare Refresh-Fenster.
- **Tier C**: explorative oder experimentelle Lasten mit strengeren Leitplanken.
- **Tier D**: serverless Zusatzdienste mit explizitem Monitoring. So wird Kostenkontrolle handhabbar, statt nur auf Monatsrechnungen zu reagieren.

7.13. Performance-Regressionen früh stoppen

Führe einen festen Regressionstest pro Release ein:

- gleiche Kernqueries,
- gleiche Warehouse-Zuordnung,
- gleiche Freshness-Annahmen,
- gleicher Blick auf Credits.

Abbruchkriterien:

- P95/P99 verschlechtern sich über Schwellwert.
- Queueing steigt signifikant.

- `bytes_scanned` explodieren ohne fachlichen Grund.
- Dynamic-Table- oder Task-Laufzeiten kippen.
- serverless Nebenkosten steigen unerklaert.

---

8. Sicherheit, Governance & Compliance

8.1. Sicherheitsgrundsatz

Eine funktionierende Snowflake-Installation ist noch keine sichere Snowflake-Plattform. Produktionsreife bedeutet mindestens:

- sauberes Rollenmodell,
- Identitaets- und Netzwerkhaertung,
- Datenmaskierung und ggf. Row Access,
- auditierbare Governance,
- kontrollierte Freigabeprozesse.

8.2. Snowflake-Zugriffskontrolle richtig lesen

Snowflake kombiniert mehrere Modelle:

- **DAC**: Objekte haben einen Owner.
- **RBAC**: Rechte werden Rollen zugewiesen, Rollen werden Nutzern zugewiesen.
- **UBAC**: direkte Nutzerzuweisungen spielen in bestimmten Konstellationen ebenfalls eine Rolle. In der Praxis sollte RBAC klar dominieren. Direktzuweisungen an Nutzer erzeugen auf Dauer fast immer Drift.

8.3. Account Roles vs. Database Roles

Snowflake unterscheidet:

- **Account Roles** für Aktionen über das gesamte Account-Scope.
- **Database Roles** für Berechtigungen innerhalb einer einzelnen Datenbank. Database Roles können nicht direkt in einer Session aktiviert werden, sondern werden an Account Roles gebunden. Das ist in grossen Umgebungen sehr hilfreich für:
- saubere Domainengrenzen,
- delegierte Verantwortlichkeit,
- besser lesbare Rollenhierarchien.

8.4. Managed Access und Ownership sauber organisieren

Ownership ist in Snowflake maechtig. Wenn Ownership, Objektanlage und operative Nutzung unklar bleiben, entstehen schnell Schatten-Admins. Praxisregeln:

- Objekt-Owner bewusst festlegen.
- Grant-Fluesse standardisieren.
- Schema- und Datenprodukt-Owner nicht dem Zufall ueberlassen.
- Managed-Access-Schemata dort nutzen, wo Freigaben zentral kontrolliert werden sollen.

8.5. Dynamic Data Masking

Dynamic Data Masking ist eine column-level security Funktion, die Werte zur Query-Zeit je Kontext und Rolle maskiert. Die Doku betont explizit:

- Policies sind Schema-Objekte.
- dieselbe Policy kann breit wiederverwendet werden.
- auch privilegierte Nutzer können dadurch gezielt eingeschränkt werden.

Das ist besonders stark für:

- PII,
- finanzielle Detailwerte,
- partnerfaehige Datenprodukte,
- Umfeld mit klarer Trennung zwischen Objektbesitz und Datensichtbarkeit.

8.6. Row Access Policies

Row-Level Security wird in Snowflake über Row Access Policies umgesetzt. Zur Query-Zeit erstellt Snowflake dabei sinngemaess eine sichere Sicht auf das Objekt und liefert nur Zeilen, deren Policy-Ausdruck zu `TRUE` evaluiert. Typische Einsatzfaelle:

- regionale Datensicht,
- Mandantentrennung,
- bereichsspezifische Zugriffsschnitte.

Wichtig:

- Row Access ist Teil des Datenprodukt- und Rollenmodells.
- Wer sie nur "nachträglich darueberlegt", bekommt häufig komplexe Sonderfälle.

8.7. Tag-Based Masking und automatische Klassifizierung

Tag-basierte Masking-Policies reduzieren manuellen Aufwand erheblich:

- Policy an Tag koppeln.
- Tag auf Datenbank, Schema, Tabelle oder Spalten anwenden.
- neue passende Objekte werden automatisch geschützt. Zusätzlich kann Snowflake sensible Daten automatisch klassifizieren und Tags zuordnen. Das ist stark für:
- grosse, dynamische Landschaften,
- Standardisierung über Teams,
- Governance-by-Default statt Governance-by-Nacharbeit.

8.8. Secure Data Sharing und Governance zusammendenken

Secure Data Sharing ist kein reines Integrationsfeature, sondern ein Governance-Thema. Fragen vor jeder Freigabe:

- Ist die Semantik des Produkts stabil?
- Sind Masking- und Row-Access-Regeln kompatibel?
- Ist klar, welche Rollen beim Provider und beim Consumer relevant sind?
- Gibt es Support- und Incident-Prozesse für den Consumer?

8.9. Governance-Bausteine jenseits von Rollen

Für reifere Umgebungen sollten mindestens mitgedacht werden:

- Tags und Klassifizierung.
- Access History und Lineage.
- Datenqualitaets- und Erwartungsmechanismen.
- Budget-/Kostenverantwortung je Domane.
- klare Aenderungs- und Freigabeprozesse.

8.10. So setzt du es um - Checkliste

- RBAC als Standard festlegen, Direktrechte auf Nutzer minimieren.
- Database Roles für domainennahe Objektgrenzen nutzen.
- Sensible Spalten über Masking-Policies absichern.
- Regionale oder mandantenbezogene Sicht über Row Access abbilden.
- Tagging und Klassifizierung für wiederkehrende Governance-Aufgaben etablieren.

8.11. Rollenrezertifizierung und Drift-Kontrolle

In vielen Umgebungen ist nicht die Erstkonfiguration das Problem, sondern Drift über Monate. Bewaehrte Praxis:

- Quartalsweise Rezertifizierung wichtiger Rollen.
- Review externer Shares und Consumer.
- Prüfung, welche Warehouses und Datenprodukte welche Rollen wirklich brauchen.
- Altlasten abbauen statt nur neue Rollen hinzuzufuegen.

8.12. Governance-Metriken für Reviews

Sicherheit und Governance werden handhabbar, wenn du sie misst. Sinnvolle Kennzahlen:

- Anteil der Datenprodukte mit dokumentiertem Rollenmodell.
- Anteil sensitiver Spalten mit aktiver Schutzpolicy.
- Anzahl externer Shares ohne aktuellen Owner-Review.
- Anzahl aktiver Warehouses ohne Budget-/Monitor-Zuordnung.
- Anteil der Core-/Serving-Objekte mit klarer Lineage und Freigabestatus.

8.13. Security-Incident-Drill

Wie bei Verfuegbarkeitsdrills sollte es auch Security- und Governance-Drills geben. Beispielszenarien:

- Falsch zugewiesene Rolle sieht zu viele Daten.
- Partner bekommt versehentlich zu breiten Share.
- Masking wurde bei Schemaaenderung nicht mehr korrekt angewendet.
- Regionale Zeilenfilter greifen nicht wie erwartet.

Ziel:

- Erkennungsgeschwindigkeit messen.
- Rollback- und Kommunikationsweg testen.
- Nachvollziehbarkeit in Access History und Governance-Artefakten prüfen.

---

9. Weiterführende Links + Uebungsaufgaben

9.1. Offizielle Kernquellen (Snowflake)

Plattform & Architektur:

- Doku-Start
- Key Concepts & Architecture
- Release Notes

Storage, Tabellen, Resilienz:

- Micro-Partitions & Data Clustering
- Automatic Clustering
- Time Travel
- Time Travel & Fail-safe
- Cloning

Laden & Transformieren:

- Snowpipe
- Streams
- Tasks
- Dynamic Tables
- Materialized Views

Performance & Kosten:

- Persisted Query Results
- Warehouse Cache
- Reducing Queues
- Search Optimization Service
- Query Acceleration Service
- Account Budget
- Resource Monitors

Security & Governance:

- Access Control Overview
- Dynamic Data Masking
- Row Access Policies
- Tag-Based Masking
- Sensitive Data Classification
- Secure Data Sharing

9.2. Uebungsaufgaben (steigende Schwierigkeit)

1. **Warehouse-Trennung (leicht)**

- Lege getrennte Warehouses für `etl`, `bi` und `sandbox` an.
- Miss, wie sich Queueing und Credit-Verbrauch verändern.

2. **Pruning-Analyse (leicht bis mittel)**

- Baue eine grosse Faktentabelle mit Datums- und Regionsfiltern.
- Vergleiche Query-Verhalten mit guter und schlechter Datenordnung.

3. **CDC-Pipeline (mittel)**

- Lade Rohdaten in eine Landing-Tabelle.
- Erstelle Stream und Task.
- Merge die Änderungen inkrementell in eine Core-Tabelle.

4. **Dynamic Table vs. Task-Kette (mittel)**

- Implementiere dieselbe Serving-Tabelle einmal mit Dynamic Table und einmal mit Streams/Tasks.
- Vergleiche Freshness, Steuerbarkeit und Kosten.

5. **Search Optimization / QAS-Vergleich (mittel bis schwer)**

- Identifiziere selektive Hot-Queries.
- Prüfe, ob Search Optimization oder QAS für den Fall sinnvoller ist.

6. **Governance-Hardening (schwer)**

- Setze Database Roles, Masking und Row Access für ein sensibles Datenprodukt um.
- Prüfe, ob unterschiedliche Rollen wirklich unterschiedliche Ergebnisse sehen.

7. **Data Sharing Drill (schwer)**

- Erstelle ein explizites Share-Produkt.
- Teste Freigabe, Clone-gestuetzte QA und einen Rollback bei Schemaaenderung.

9.3. Abnahmekriterien für "produktionsreif"

- Kernworkloads haben eigene oder bewusst geteilte Warehouse-Strategien.
- Freshness-Ziele sind je Datenprodukt definiert und messbar.
- Kostenpfade für Warehouses und serverless Features sind sichtbar.
- Mindestens ein Governance-Layer (Masking/Row Access/Tags) ist wirksam eingebaut.
- Release-, Clone- und Incident-Prozesse sind für kritische Datenprodukte geprobt.

9.4. Abschluss: Was "Pro-Level Snowflake" in der Praxis bedeutet

Pro-Level heisst nicht, jede neue Funktion sofort zu aktivieren. Pro-Level heisst:

- Du trennst Storage-, Compute- und Governance-Entscheidungen sauber.
- Du modellierst Datenprodukte statt nur Tabellen.
- Du behandelst Kosten als Teil des Betriebs.
- Du optimierst zuerst Scans und Workload-Isolierung, nicht reflexhaft Warehouse-Groesse.
- Du machst Sharing und Sicherheit zu expliziten Architekturentscheidungen. Wenn diese Punkte stehen, wird Snowflake nicht nur schnell, sondern organisatorisch belastbar.

9.5. So setzt du es um - Checkliste

- Alle offiziellen Quellen im Team-Wiki als Primaerreferenz hinterlegt.
- Für jedes kritische Datenprodukt existiert ein Owner, ein Warehouse-Modell und ein Governance-Profil.

- Kostenbudgets und Warehouse-Monitore sind eingerichtet.
- Uebungen 1 bis 7 mit messbaren Ergebnissen dokumentiert.
- Architektur-, Betriebs- und Security-Review gemeinsam abgeschlossen.

9.6. 90-Tage-Umsetzungsfahrplan

Tage 1-30

- Workloads und Datenprodukte inventarisieren.
- Warehouse-Trennung und Kostenbaseline aufbauen.
- Raw/Core/Serving sauber definieren. **Tage 31-60**
- Inkrementelle Pipelines stabilisieren.
- Query- und Queueing-Monitoring institutionalisieren.
- Governance-Layer für sensible Datenprodukte aktivieren. **Tage 61-90**
- Kostenleitplanken für serverless Features schaerfen.
- Shares, Clones und Resilienzpfade testen.
- Performance-Regressionstest und Review-Rhythmus etablieren.

9.7. Self-Assessment für Teams

Bewerte euer Team ehrlich mit 0 bis 2 Punkten je Aussage:

- Wir haben klare Workload-Grenzen und Warehouse-Owner.
- Wir kennen unsere teuersten Queries und deren Hauptursachen.
- Wir unterscheiden bewusst zwischen View, Materialized View und Dynamic Table.
- Wir sehen Warehouses und serverless Kosten getrennt.
- Unsere sensiblen Datenprodukte haben funktionierende Schutzmechanismen.
- Shares und externe Consumer sind fachlich und technisch dokumentiert.
- Unsere Alerts führen zu konkreten nächsten Schritten.

Auswertung:

- 0-5 Punkte: frühe Plattformphase, hohe Betriebs- und Kostenrisiken.
- 6-10 Punkte: solide Basis, aber inkonsistente Steuerung.
- 11-14 Punkte: gutes Produktionsniveau.

9.8. Nächste Ausbaustufen

Wenn die Basis stabil steht, sind sinnvolle nächste Ausbaustufen:

- Datenqualitaets- und Erwartungspruefungen je Kernprodukt.
- Share-Katalog mit expliziten Produktversionen.
- Automatisierte Kostenanomalie-Reviews.
- Regelmäßige Bewertung von Search Optimization, Clustering und Materialized Views gegen reale Nutzung.
- Quartalsweiser "Snowflake Health Review", der Architektur, Kosten, Governance und Nutzerwirkung gemeinsam betrachtet.

Über den Autor

Erhard Rainer berät kleine und mittlere Unternehmen in Österreich zu Business Intelligence, Machine Learning und KI-Agenten. Er hat 15 Jahre Erfahrung in Data Analytics, davon 10 Jahre in der Leitung von Teams und Departments in Konzernen, und ist Dozent für betriebswirtschaftliche und Data-Analytics-Fächer.

Sprechen Sie mit mir über Ihre Daten: Das Erstgespräch ist kostenlos.

https://www.thinkbigger.at/contact.html#anfrage

© 2026 Think BIGGER – Erhard Rainer. Dieses PDF gibt den Stand des Online-Artikels vom 06.07.2026 wieder; maßgeblich ist die Online-Fassung unter https://www.thinkbigger.at/wissen/snowflake/