Think BIGGER

DEEP DIVE FÜR FACHLEUTE

BigQuery

Autor	Erhard Rainer
Veröffentlicht	15.07.2019
Aktualisiert	11.04.2026
Umfang	ca. 35 Min. Lesezeit
Online	www.thinkbigger.at/wissen/bigquery/

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 BigQuery die richtige Wahl ist 4
2. Architektur & Kernkonzepte 7
3. Datenmodellierung in der Praxis 10
4. GoogleSQL - systematische Praxisreferenz 14
5. Datenaufnahme, Transformation & Delivery 17
6. Betrieb, Beobachtbarkeit & Troubleshooting 20
7. Performance, Skalierung & Kostenkontrolle 22
8. Sicherheit, Governance & Compliance 25
9. Weiterführende Links + Übungsaufgaben 28

0. Überblick (für Eilige)

BigQuery ist ein serverloses, spaltenorientiertes Analytics-System von Google Cloud für Data Warehousing, BI, ELT, Streaming-Analytics und zunehmend auch Such-, Vektor- und ML-nahe Workloads. Diese Seite ist eine praxisorientierte Referenz für Architektur, Datenmodellierung, GoogleSQL, Ingestion, Betrieb, Performance, Kostenkontrolle und Sicherheit. Ziel ist nicht nur "was ist BigQuery", sondern "wie setze ich BigQuery so ein, dass Abfragen schnell, Kosten kontrolliert und Governance sauber bleiben".

> Hinweis zum Produktstand: Diese Referenz ist fachlich auf Stand **11. April 2026**. BigQuery ist ein gemanagter Dienst ohne klassische Major-/Minor-Versionen. Versionssensitive Aussagen beziehen sich daher auf die zu diesem Zeitpunkt öffentlich dokumentierten BigQuery-Funktionen wie Editions, Reservations, Storage Write API, Partitionierung, Search/Vector Indexes sowie Row-/Column-Level Security.

0.1. Inhaltsverzeichnis (kurz)

- 0. Überblick (für Eilige)
- 1. Einordnung: Wann BigQuery die richtige Wahl ist
- 2. Architektur & Kernkonzepte
- 3. Datenmodellierung in der Praxis
- 4. GoogleSQL - systematische Praxisreferenz
- 5. Datenaufnahme, Transformation & Delivery
- 6. Betrieb, Beobachtbarkeit & Troubleshooting
- 7. Performance, Skalierung & Kostenkontrolle
- 8. Sicherheit, Governance & Compliance
- 9. Weiterführende Links + Übungsaufgaben

0.2. Quick-Glossar

- **Dataset**: Logischer Container für Tabellen, Views, Routinen und Zugriffsgrenzen.
- **Slot**: Recheneinheit für Query- und Hintergrund-Workloads in reservations-/editions-basierten Setups.
- **Reservation**: Dedizierter Slot-Pool für definierte Workloads.
- **Assignment**: Zuordnung von Projekten, Ordnern oder Organisationseinheiten zu Reservations.
- **Partitionierung**: Aufteilung grosser Tabellen in physisch separierte Datenbereiche, typischerweise nach Datum/Zeit.
- **Clustering**: Sortierung von Daten innerhalb einer Tabelle oder Partition nach ausgewählten Spalten zur Block-Pruning-Optimierung.
- **Materialized View**: Physisch gepflegte, voraggregierte oder vorab berechnete Sicht.
- **Authorized View**: View, die Daten gezielt freigibt, ohne den Zugriff auf das Quelldataset direkt zu öffnen.
- **Policy Tags**: Taxonomie-basierte Kennzeichnungen für Column-Level Security.
- **Storage Write API**: Empfohlene API für Streaming-Ingestion mit at-least-once oder exactly-once Semantik je Pattern.
- **BI Engine**: In-Memory-Beschleunigung für bestimmte BI- und Dashboard-Abfragen.
- **Time Travel**: Rückgriff auf frühere Tabellenversionen innerhalb des konfigurierten Fensters.

0.3. Lernpfad: Vom Einstieg zur Produktion

1. Verstehe zuerst das Betriebsmodell: Storage vs. Compute, Slots, Jobs, Partitionierung, Clustering.
2. Modelliere danach Tabellen entlang echter Fragestellungen, nicht entlang eines reinen Quellschemas.
3. Baue dann GoogleSQL-Pfade für Ingestion, Transformation und Auslieferung reproduzierbar auf.
4. Führe Kosten-, Performance- und Sicherheitsleitplanken ein, bevor Self-Service breit ausgerollt wird.
5. Sichere zuletzt den Betrieb mit Runbooks, Monitoring, Row-/Column-Level Controls und Restore-Strategien ab.

0.4. Mini-Projekt (durchgehend): Commerce-Analytics-Plattform

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

- Ein internationaler Haendler betreibt Webshop, Mobile App und stationaeren Vertrieb.
- Pro Tag fallen rund 1,2 Milliarden Event-Zeilen, mehrere Millionen Bestellungen und tägliche ERP-/CRM-Snapshots an.
- Dashboards für Vertrieb und Marketing sollen im 5-Minuten-Takt aktualisiert werden.
- Finanzzahlen müssen täglich nachvollziehbar, revisionsfaehig und regional getrennt auswertbar sein.
- PII darf nur für klar definierte Rollen sichtbar sein.
- Das Controlling verlangt klare Kostenobergrenzen und verursachungsgerechte Zuordnung.

Wir verwenden dieses Szenario fortlaufend für:

- Tabellen- und Layer-Design.
- Partitionierungs- und Clustering-Entscheidungen.
- Streaming- vs. Batch-Ingestion.
- Reservations- und Kostensteuerung.
- Governance- und Security-Mechanismen.

0.5. Profi-Heuristiken

- BigQuery ist stark, wenn du grosse Datenmengen analytisch, parallel und SQL-nah auswerten willst.
- BigQuery ist kein Ersatz für eine transaktionsoptimierte OLTP-Datenbank.
- Storage ist oft guenstig, unkontrollierter Scan und Slot-Verbrauch sind teuer.
- `SELECT *` auf grossen Tabellen ist kein Komfort, sondern häufig ein Budgetfehler.
- Partitionierung ohne disziplinierte Filterlogik spart nicht automatisch Geld.
- Wenige, gut gewaehle Cluster-Spalten schlagen viele halbpassende.
- Verschachtelte und wiederholte Felder können Joins deutlich reduzieren.
- Workload-Isolation über Reservations ist oft wichtiger als das letzte Prozent SQL-Tuning.
- Governance muss im Modell angelegt sein, nicht erst im Audit nachgezogen werden.

0.6. So setzt du es um - Checkliste

- Definiere 5 bis 10 Kernfragen, Reports oder API-nahen Analysepfade vor dem Modellieren.
- Lege pro Workload Zielwerte für Latenz, Aktualitaet und maximale Kosten fest.
- Entscheide bewusst zwischen On-Demand und Editions-/Reservation-Modell.
- Erzwinge für grosse Fakttabellen Partitionierungs- und Pruning-Disziplin.
- Trenne mindestens Prod, Dev/Test und Ad-hoc-Analyse rechnerisch oder organisatorisch.
- Lege ein Restore-/Rollback-Muster mit Time Travel, Snapshots oder Clones fest.

0.7. Mentales Modell für den Alltag

Wenn du BigQuery in Projekten sicher einsetzen willst, hilft ein klares mentales Modell: BigQuery ist kein "riesiges SQL-PostgreSQL in der Cloud", sondern eine Kombination aus gemanagtem Columnar Storage, massiv paralleler Ausführung, jobbasiertem Betriebsmodell und zentraler Governance. Viele Fehlentscheidungen entstehen, weil Teams nur auf eine Ebene schauen. Analysten sehen die SQL-Flexibilitaet, Plattformteams sehen den serverlosen Komfort, Finance sieht nur die Rechnung, und Security betrachtet IAM sowie Policy Tags. Produktionsreife entsteht aber erst dann, wenn diese Ebenen in einem gemeinsamen Betriebsvertrag zusammenlaufen: Welche Abfragen sind first-class? Welche Tabellen sind kostenkritisch? Welche Workloads dürfen konkurrieren? Welche Daten müssen regional getrennt bleiben? Welche Rollen dürfen Rohdaten sehen? Die entscheidende Leitfrage pro Architekturentscheidung lautet deshalb: "Welches messbare Endverhalten bei Kosten, Latenz, Aktualitaet oder Datenzugriff verbessert sich?" Wenn darauf niemand konkret antworten kann, ist die Entscheidung meist nur implizite Hoffnung. Für die tägliche Arbeit bedeutet das: Tabellen werden nicht nur nach Quellsystem, sondern nach Analyse- und Governance-Zweck gebaut; jedes neue Dashboard bekommt ein Kosten- und Aktualitaetsprofil; jede Materialisierung bekommt eine Begründung; jede Reservation bekommt einen Owner; und jede sensible Spalte bekommt ein Zugriffsmodell. Teams, die dieses Modell leben, diskutieren weniger über "irgendwie läuft die Query langsamer" und mehr über pruefbare Zusammenhänge wie Partition-Pruning, Slot-Contention, Join-Skew, Refresh-Fenster und Zugriffspfade. Genau das macht aus BigQuery kein Tool für schnelle Demos, sondern eine belastbare Analyseplattform.

---

1. Einordnung: Wann BigQuery die richtige Wahl ist

1.1. Problemklassen, in denen BigQuery typischerweise stark ist

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

- Sehr grosse Datenmengen müssen analytisch ausgewertet werden.
- SQL soll das primäre Arbeitsmittel für Analysten, Analytics Engineers und Data Teams sein.
- Die Infrastruktur für Storage, Skalierung und Query-Parallelisierung soll nicht selbst betrieben werden.
- Mehrere Teams brauchen Self-Service-Zugriff auf gemeinsame Datenplattformen.
- Batch und Streaming sollen in einer gemeinsamen Analyseumgebung zusammenlaufen.

Typische Domaenen:

- Data Warehousing und Reporting.
- Marketing-, Produkt- und Event-Analytics.
- IoT- und Log-Analytics.
- ELT-Plattformen für Roh-, Kern- und Mart-Schichten.
- ML-nahe Datavorbereitung, Features und Auswertung.

1.2. Wann ein OLTP-System oder anderes System oft besser passt

Ein anderes System passt meist besser, wenn du primär brauchst:

- Millisekunden-schnelle Punktupdates auf wenige Zeilen.
- Sehr hohe Frequenz kleiner Transaktionen.
- Strenge, interaktive Row-by-Row-Workflows.
- Session-zentrierte Anwendungen mit vielen gleichzeitigen Schreibvorgaengen.

Typische bessere Kandidaten in solchen Fällen:

- PostgreSQL oder MySQL für klassische OLTP-Anwendungen.
- Spanner oder CockroachDB für global verteilte, transaktionale Systeme.
- Pub/Sub + operational store für eventgetriebene Online-Pfade.

1.3. BigQuery vs. klassisches Data Warehouse

BigQuery reduziert gegenüber klassisch selbst betriebenen Warehouses viele Betriebsaufgaben:

- Kein Cluster-Patching.
- Kein manuelles Storage-Tiering.
- Keine Node-Kapazitaetsplanung im alten Sinn.
- Kein klassischer Index-Wildwuchs wie in traditionellen RDBMS-Systemen.

Dafür verschieben sich die Hauptfragen:

- Wie strukturiere ich Tabellen für Pruning und geringe Scan-Kosten?
- Wie isoliere ich konkurrierende Workloads?
- Wie halte ich Self-Service unter Kosten- und Governance-Kontrolle?

1.4. Wann BigQuery gegenüber Lakehouse-/Datei-zentrierten Setups punktet

BigQuery ist häufig stark, wenn du willst:

- Weniger Betriebsaufwand als bei stark selbst orchestrierten Lakehouse-Stacks.
- Sofort nutzbare SQL- und Governance-Funktionen.
- Gute Integration in Google-Cloud-Workflows.
- Einheitliche Plattform für BI, Data Engineering, Security und teilweise ML.

Lakehouse-/Datei-zentrierte Setups können attraktiver sein, wenn:

- Offene Dateiformate als primäre Wahrheit strikt vorgeschrieben sind.
- Eine sehr tiefe Engine-Kontrolle oder Multi-Engine-Strategie im Vordergrund steht.
- Rechen- und Speicherebenen über verschiedene Plattformen sehr bewusst getrennt werden sollen.

1.5. Harte No-Go-Faelle für BigQuery

- Das Hauptsystem ist ein transaktionaler Online-Store mit sehr häufigen Punktupdates.
- Das Team erwartet, dass alle Abfragen ohne Modellierungsdisziplin automatisch billig und schnell sind.
- Datenresidenz und Dataset-Locations sind fachlich kritisch, aber organisatorisch ungeklärt.
- Es gibt kein Governance-Modell für sensible Spalten, Rollen und Freigaben.

> **Anti-Pattern:** "Wir laden erstmal alles roh in BigQuery und schauen später, wie wir es strukturieren." Ergebnis in der Praxis: teure Vollscans, inkonsistente Definitionen, Sicherheitsluecken und späte Refactorings unter Produktivdruck.

1.6. Entscheidungsmatrix (kompakt)

Kriterium	BigQuery passt	BigQuery passt eher nicht
Primärer Workload	Analytik, BI, ELT, Exploration	OLTP, Chatty-Transaktionen
Datenvolumen	Gross bis sehr gross	Klein und transaktionslastig
Bedienmodell	SQL, Jobs, Self-Service	Zeilenweise App-Interaktion
Skalierung	Serverlos, workload-basiert	Feinsteuerung pro Node/Instance gewuenscht
Governance	Zentral über IAM, Views, Policy Tags	Fachlich kaum geklärt
Kostenmodell	Scan-/Slot-getrieben steuerbar	Unkontrollierte Ad-hoc-Nutzung erwartet

1.7. So setzt du es um - Checkliste

- Formuliere drei reale Analysefragen mit echten Datenmengen und echten Aktualitaetszielen.
- Vergleiche BigQuery mindestens mit einer OLTP- und einer alternativen Analytics-Option.
- Dokumentiere explizit, welche Workloads **nicht** in BigQuery landen sollen.
- Lege fest, welche Teams Daten lesen, schreiben, freigeben und absichern dürfen.
- Definiere ein Exit-Kriterium: Wann wäre BigQuery für diesen Anwendungsfall die falsche Wahl?

1.8. Praxisvergleich: BigQuery vs. "wir optimieren später"

Der häufigste Einfuehrungsfehler ist nicht fehlendes SQL-Wissen, sondern ein falscher Projektmodus. Viele Teams starten mit:

- Rohdaten laden wir erstmal vollständig.
- Partitionierung denken wir später nach.
- Views und Marts bauen wir, wenn die Dashboards langsam werden.
- Kosten analysieren wir nach dem ersten Monatsreport.

In BigQuery führt diese Reihenfolge oft zu:

- Dauerhaften Vollscans.
- Zu spät erkannter Workload-Konkurrenz.
- Unscharfen Semantiken zwischen Roh-, Kern- und Reportingschicht.
- Schwer aufraeumbaren Berechtigungsstrukturen.

Die bessere Reihenfolge ist:

1. Business-Fragen und Aktualitaetsfenster definieren.
2. Tabellen- und Layer-Design festlegen.
3. Security- und Kostenleitplanken einziehen.
4. Erst dann Self-Service breit freigeben.

1.9. Architekturentscheidungen sauber dokumentieren

Bewaehrte ADR-Fragen (Architecture Decision Record):

- Warum BigQuery und nicht Alternative X?
- Welche Tabellen sind Rohdaten, welche sind harmonisierte Kerndaten, welche sind Marts?
- Welche Workloads laufen On-Demand, welche über Reservations?
- Welche Daten dürfen welche Region nicht verlassen?

- Welche sensiblen Spalten werden per Policy Tag, Masking, Authorized View oder Separierung kontrolliert?

Ergaenze jede ADR um "Betriebsfolgen":

- Welche Metriken zeigen früh, dass die Entscheidung kippt?
- Welcher Query-Typ dürfte als Erstes zu teuer werden?
- Welcher Incident-Pfad ist denkbar?
- Welche Rueckfallstrategie existiert?

---

2. Architektur & Kernkonzepte

2.1. Serverloses Columnar-Analytics-Modell

BigQuery trennt Storage und Compute logisch voneinander. Für die Praxis bedeutet das:

- Daten liegen spaltenorientiert und hochgradig parallel auswertbar vor.
- Rechenkapazitaet wird pro Job bzw. Workload bereitgestellt.
- Du planst nicht primär Nodes, sondern Kosten-, Slot- und Workload-Verhalten.

Das verändert die Architekturfrage von "wie gross ist mein Cluster?" zu:

- Wie viel wird gelesen?
- Wie viel konkurriert gleichzeitig?
- Wie stark ist mein Modell auf Pruning und Preaggregation optimiert?

2.2. Ressourcenmodell: Projekt, Dataset, Tabelle, View, Routine, Job

Das BigQuery-Ressourcenmodell ist operativ zentral:

- **Projekt**: technische und finanzielle Abrechnungsgrenze, IAM-Kontext.
- **Dataset**: logischer Container, oft Sicherheits- und Ownership-Grenze.
- **Tabelle**: Standardobjekt für native oder externe Daten.
- **View / Materialized View**: Abstraktion bzw. Beschleunigung für Abfragen.
- **Routine**: UDF, Table Function, Stored Procedure.
- **Job**: Query, Load, Copy, Extract oder Hintergrundaktivitaet.

Viele Architekturprobleme sind eigentlich Grenzprobleme:

- Dataset falsch geschnitten.
- Projektkosten nicht sauber trennbar.
- Jobs konkurrieren ohne klare Verantwortlichkeit.

2.3. Location-Strategie und Datenresidenz

Locations sind kein Spaetthema, sondern Grundsatzentscheidung. Du entscheidest früh:

- Region vs. Multi-Region.
- EU- vs. US-Trennung.
- Wo Rohdaten, Kernschicht und Marts liegen.

Praktische Folgen:

- Dataset-Locations beeinflussen Kopier-, Join- und Governance-Pfade.
- Policy-Taxonomien und Datenobjekte müssen regional sauber zueinander passen.
- Standortfehler werden oft erst spät sichtbar, wenn Pipelines, Kosten oder Compliance bereits daran hängen.

2.4. Compute-Modelle: On-Demand vs. Editions

BigQuery bietet zwei Grundlogiken für Compute:

- **On-Demand**: Abrechnung nach verarbeiteten Datenmengen, gut für variable oder sporadische Last.
- **Editions/Reservations**: Slot-basierte Kapazitaet, gut für planbare, wiederkehrende oder strikt zu isolierende Workloads.

Praxisregel:

- On-Demand ist oft gut für Exploration, kleine Teams oder unregelmaessige Last.
- Reservations sind oft besser für geregelten Produktivbetrieb, gemischte Teams und Kosten-/SLA-Steuerung.

Wichtig:

- BigQuery Editions sind eine Eigenschaft der Compute-Schicht, nicht des Storage.
- Projekte können in der Praxis je nach Setup gemischte Modelle nutzen.

2.5. Slots, Reservations, Assignments und BI Engine

Slots sind die operative Recheneinheit. Reservations und Assignments helfen dir, Rechenbudget bewusst zu verteilen:

- Produktions-Dashboards sollen nicht mit Ad-hoc-Analysen konkurrieren.
- Hintergrund-Jobs sollen interaktive Reports nicht blockieren.
- Testprojekte sollen nicht versehentlich Produktivkapazitaet verbrennen.

BI Engine ist davon getrennt zu betrachten:

- Es ist eine In-Memory-Beschleunigung für bestimmte BI-Abfragen.
- BI-Acceleration ersetzt keine saubere Tabellen- und Query-Architektur.

2.6. Tabellentypen: Native, externe und BigLake-nahe Muster

Aus Architektursicht sind drei Denkweisen wichtig:

- **Native BigQuery-Tabellen** für stabile, performante Analytics.
- **Externe Tabellen** für Daten, die ausserhalb von BigQuery verbleiben.
- **BigLake-/Hybridmuster** für governte Zugriffe auf externe Daten mit einheitlicherem Management.

Heuristik:

- Wenn ein Datensatz häufig, geschaeftskritisch und latenzsensibel abgefragt wird, sind native Tabellen meist die robusteste Wahl.
- Externe Muster sind stark für Dateibestaende, geteilte Datenseen oder schrittweise Migrationen.

2.7. Query- und Job-Lebenszyklus

BigQuery arbeitet jobbasiert. Wichtige Jobtypen:

- Query Jobs.
- Load Jobs.
- Copy Jobs.
- Extract Jobs. Jeder Job hinterlaesst Metadaten, die für Betrieb und Kostensteuerung wertvoll sind:
- `total_bytes_processed`
- `total_slot_ms`
- `statement_type`
- Fehler-, Timing- und Planinformationen Die Job-Ebene ist deshalb nicht nur Debug-Hilfe, sondern Kern der Plattformbeobachtung.

2.8. Time Travel, Snapshots und Clones

BigQuery bietet mehrere Rueckfallebenen:

- **Time Travel** für Rückgriff auf frühere Zustaende innerhalb des konfigurierten Fensters.
- **Snapshots** für read-only, klar referenzierbare Stichtagsabbilder.
- **Clones** für leichte, schnell erstellbare Arbeitskopien.

Das ist operativ wertvoll für:

- Fehlerhafte Loads.
- Falsche `MERGE`-Statements.
- Test- und Vergleichsszenarien.
- Revisionsnahe Analysen.

2.9. Mini-Projekt: Architekturentscheidung

Für die Commerce-Plattform aus Abschnitt 0:

- Rohdaten landen batchweise als Parquet in Cloud Storage und werden anschliessend geladen.
- Near-Real-Time-Events kommen über die Storage Write API.
- Es gibt getrennte Datasets für `raw`, `core`, `mart` und `sandbox`.
- Produktive Dashboards und ETL-Hintergrundjobs erhalten getrennte Reservations/Assignments.
- PII-relevante Daten liegen in klar abgegrenzten Datasets mit feingranularen Policies.

2.10. So setzt du es um - Checkliste

- Lege Dataset-Locations vor dem ersten produktiven Load fest.
- Dokumentiere pro Workload das Compute-Modell und den Owner.
- Trenne interactive, batch und experimentelle Last bewusst.
- Entscheide pro Datendomäne bewusst zwischen nativer und externer Speicherung.
- Definiere ein Restore-Muster mit Time Travel, Snapshot oder Clone.
- Führe ein technisches Inventar aller produktiven Datasets, Reservations und kritischen Views.

2.11. Architektur-Risiken früh erkennen

Typische Fruehwarnzeichen:

- Dashboards werden "zufaellig" langsam, wenn Backfills laufen.
- Gleichartige Abfragen schwanken stark in Laufzeit und Kosten.
- Mehrere Projekte teilen sich stillschweigend denselben Rechenpool.

- Sensible Tabellen werden über ein allgemeines Analysten-Dataset exponiert.
- Regionale Trennungen sind nur in Namenskonventionen, nicht im Ressourcenmodell sichtbar. Diese Signale zeigen oft kein einzelnes Bugproblem, sondern eine unklare Plattformarchitektur.

2.12. Architektur-Check vor Produktion

Vor produktivem Traffic sollte mindestens geprüft sein:

- Dataset-Locations und Datenresidenz sind fachlich und technisch abgestimmt.
- Workload-Gruppen sind identifiziert und, falls nötig, rechnerisch getrennt.
- Kritische Tabellen sind nativer oder bewusst begruendeter externer Typ.
- Restore-Szenarien sind getestet.
- Kosten-Owner und Security-Owner sind explizit benannt.
- Monitoring deckt Job-, Kosten- und Sicherheitsereignisse ab.

2.13. Architektur-Entscheidungen an SLOs koppeln

Architektur wird erst belastbar, wenn sie an Zielwerte gekoppelt ist. Beispiele:

- SLO: Dashboard-Aktualitaet < 5 Minuten.
- Konsequenz: Streaming oder häufige Micro-Batches plus klare Reservation für das Semantic Layer.
- SLO: Tagesabschluss bis 06:00 Uhr.
- Konsequenz: Hintergrund-Reservation, Backfill-Fenster, Snapshot vor Endverarbeitung.
- SLO: PII nur für Finance und Fraud.
- Konsequenz: Policy Tags, Authorized Views und dataset-scharfe Ownership.

---

3. Datenmodellierung in der Praxis

3.1. Das wichtigste Prinzip: Modellierung folgt Fragen, Kosten und Governance

In BigQuery ist die Frage nicht nur: "Wie bilde ich die Quelle technisch sauber ab?" Sondern: "Welche Abfragen müssen schnell, guenstig und sicher laufen?" Ein gutes Modell beantwortet gleichzeitig:

- Welche Tabellen werden oft und von wem gelesen?
- Welche Filter müssen Bytes sparen?
- Welche Felder sind sensibel?
- Welche Layer sind roh, stabil oder konsumierbar?

3.2. Partitionierung: Der größte Hebel für Scan-Kosten

Partitionierung ist meist der wichtigste einzelne Modellierungshebel. Typische Muster:

- `DATE(event_ts)` oder separates `event_date`.
- `order_date` für transaktionsnahe Fakttabellen.
- Ingestion-Time für generische Landing-Tabellen.
- Integer-Range bei geeigneten Fachschluesseln.

Praxisregeln:

- Partitioniere entlang häufiger Filter und fachlicher Lebenszyklen.

- Nutze `require_partition_filter`, wenn Vollscans verhindert werden sollen.
- Partitioniere nicht automatisch jede Tabelle; kleine Dimensionen brauchen es oft nicht.

3.3. Clustering: Der zweite grosse Hebel

Clustering sortiert Daten innerhalb der Tabelle oder Partition. Es ist besonders wirksam, wenn:

- wenige, hochselektive Spalten häufig in Filtern vorkommen,
- Daten gross genug sind,
- Partitionierung allein noch zu grob wäre.

Gute Kandidaten sind oft:

- `tenant_id`
- `customer_id`
- `country_code`
- häufige Lookup- oder Segmentierungsschluessel

Schlechte Kandidaten sind oft:

- niedrig kardinale Flags ohne starke Filterwirkung
- Spalten, die selten in Filtern vorkommen
- unklare "wir könnten sie mal brauchen"-Entscheidungen

3.4. Nested und Repeated Fields statt unnötiger Join-Kaskaden

BigQuery unterstützt verschachtelte (`STRUCT`) und wiederholte (`ARRAY`) Felder sehr gut. Das ist in der Praxis wichtig, weil:

- Hierarchische Daten oft gemeinsam abgefragt werden.
- Joins über grosse Faktmengen teuer und langsam werden können.
- Denormalisierung in einem Analytics-System oft sinnvoller ist als perfekte relationale Reinheit.

Beispiele:

- Bestellkopf mit `line_items` als `ARRAY<STRUCT<...>>`
- Event mit Liste technischer Parameter oder Produkte im Warenkorb

3.5. Vermeide Date-Sharding und Wildcard-Dauerarchitektur

Ein klassisches BigQuery-Anti-Pattern sind dateibenannte oder datumsnamige Tabellen wie:

- `events_20260101`
- `events_20260102`
- `events_20260103`

Warum das problematisch ist:

- Mehr Metadaten-Overhead.
- Mehr Berechtigungs- und Verwaltungsaufwand.
- Schlechtere Query-Performance als bei sauber partitionierten Tabellen.

Heuristik:

- Wenn die Daten fachlich eine Tabelle sind, sollen sie meist auch technisch eine Tabelle mit Partitionierung sein.

3.6. Layering: Raw, Core, Mart statt Monolith-Tabelle

Ein robustes Warehouse hat typischerweise mindestens drei Schichten:

- **Raw**: nahe an Quelle, auditierbar, technisch ingestiert.
- **Core**: bereinigt, harmonisiert, business-stabil.
- **Mart**: konsumorientiert für Fachbereiche, Reports oder Domänen.

Vorteile:

- Rohdaten bleiben nachvollziehbar.
- Kernlogik wird zentral wiederverwendbar.
- Fachsichten werden explizit und governance-faehig.

3.7. Materialized Views, Aggregationstabellen und Semantic-Layer-Denken

Nicht jede häufig genutzte Abfrage sollte jedes Mal neu gerechnet werden. Geeignete Werkzeuge:

- Materialized Views für passende, wiederkehrende Query-Muster.
- Aggregationstabellen für Dashboards und SLO-kritische Kennzahlen.
- Authorized Views für gezielte Freigabe.

Wichtig:

- Materialisierung ist kein Ersatz für schlechtes Rohmodell.
- Jede Materialisierung braucht Owner, Refresh-Verstaendnis und fachliche Verantwortung.

3.8. Aufbewahrung, Expiration und Datenlebenszyklus

Modellierung umfasst auch Datenlebensdauer. Wichtige Muster:

- Partition Expiration für kurzlebige Detaildaten.
- Separate Langzeit-Aggregationen für langfristige KPIs.
- Archivierung nach Cloud Storage für selten benötigte Daten.

So vermeidest du:

- Unnötige Storage-Kosten.
- Unnötige Scan-Kosten auf jahralten Rohdaten.
- Governance-Drift durch "wir löschen später mal".

3.9. Mini-Projekt: Tabellenentwurf

Ein möglicher Kernentwurf:

```
CREATE TABLE `analytics_core.fact_orders`
(
  order_id STRING,
  order_ts TIMESTAMP,
  order_date DATE,
  tenant_id STRING,
  customer_id STRING,
  country_code STRING,
  channel STRING,
```

```
  status STRING,
  total_amount NUMERIC,
  line_items ARRAY<STRUCT<
    sku STRING,
    qty INT64,
    net_amount NUMERIC
  >>
)
PARTITION BY order_date
CLUSTER BY tenant_id, country_code, customer_id
OPTIONS (
  require_partition_filter = TRUE
);
```

Warum dieses Muster?

- `order_date` erlaubt starke Pruning-Effekte.
- `tenant_id` und `country_code` passen zu häufigen Governance- und Fachfiltern.
- `line_items` vermeidet für viele Standardauswertungen einen separaten, grossen Join.

3.10. So setzt du es um - Checkliste

- Definiere pro grosse Tabelle Partitionierungsstrategie und Begründung.
- Waehle nur Cluster-Spalten mit echtem Filter- oder Sortierwert.
- Vermeide dateishardierte Tabellen für Daten, die logisch zusammengehoeren.
- Nutze Nested/Repeated Fields dort, wo Hierarchien gemeinsam gelesen werden.
- Dokumentiere Layergrenzen zwischen Raw, Core und Mart.
- Lege pro Tabelle Aufbewahrungs- und Expiration-Regeln fest.

3.11. Häufige Anti-Pattern

- Eine einzige Super-Tabelle für alle Eventtypen.
- `SELECT *` als Standardkonsumweg.
- Kein `require_partition_filter` auf teuren Fakttabellen.
- Zu viele fast identische Marts ohne zentrale Kernlogik.
- Sensible Spalten in breit verteilten, allgemeinen Analysten-Views.
- Wildcard-Abfragen über jahreslange Tabellen-Shards als Dauerloesung.

3.12. Schema-Evolution bewusst behandeln

BigQuery ist flexibel, aber Schema-Evolution muss trotzdem gestaltet werden. Praxisregeln:

- Neue Spalten sind oft einfach, aber fachlich nicht neutral.
- Umbenennungen und Typaenderungen brauchen Migrationspfade.
- JSON-Felder sind hilfreich, aber kein Ausweg aus Datenmodellverantwortung.
- Breche Änderungen erst dann aus, wenn Views, Marts und Downstream-Jobs geprüft sind.

3.13. Modellierungsentscheidungen messbar machen

Zu jeder grossen Tabelle sollten mindestens dokumentiert sein:

- Zweck und Hauptverbraucher.
- Erwartetes Tagesvolumen.
- Primäre Filter- und Join-Pfade.

- Partitionierungs- und Clustering-Begruendung.
- Sensible Spalten und Security-Mechanismus.
- Aufbewahrungsfrist und Archivmuster.

---

4. GoogleSQL - systematische Praxisreferenz

4.1. GoogleSQL ist der Standard

Für neue Lösungen sollte GoogleSQL der Standard sein. Wichtige Gründe:

- DDL, DML, TCL und moderne Funktionen liegen im GoogleSQL-Pfad.
- Legacy SQL ist primär Rueckwaertskompatibilitaet.
- Viele moderne Features wie Row-Level Security, DDL für Indexes oder Materialized Views setzen GoogleSQL voraus.

4.2. DDL: Tabellen, Views, Routinen, Indexes

GoogleSQL deckt weit mehr als `SELECT` ab:

- `CREATE TABLE`
- `CREATE VIEW`
- `CREATE MATERIALIZED VIEW`
- `CREATE FUNCTION`
- `CREATE PROCEDURE`
- `CREATE SEARCH INDEX` Das ist in BigQuery wichtig, weil Plattformarbeit nicht nur aus Query-Text besteht, sondern aus reproduzierbarer Objektverwaltung.

4.3. Projektion und Scan-Disziplin

Der häufigste Performance- und Kostenfehler ist ueberfluessige Projektion. Praxisregeln:

- Lies nur benötigte Spalten.
- Nutze `SELECT * EXCEPT (...)` beim Explorieren.
- Materialisiere grosse Zwischenmengen lieber bewusst, statt sie implizit mehrfach zu scannen.

4.4. Partition Pruning explizit schreiben

Eine gute Query erkennt man häufig daran, dass ihr Filter zur Partitionierung passt. Beispiel:

```
SELECT
  order_date,
  country_code,
  SUM(total_amount) AS revenue
FROM `analytics_core.fact_orders`
WHERE order_date BETWEEN DATE '2026-04-01' AND DATE '2026-04-07'
GROUP BY order_date, country_code;
```

Wichtig ist hier nicht nur das Ergebnis, sondern die Betriebswirkung:

- Weniger gescannte Daten.
- Niedrigere Kosten.
- Besser vorhersagbare Laufzeit.

4.5. UNNEST für verschachtelte Daten sauber nutzen

Bei Arrays brauchst du `UNNEST`. Beispiel:

```
SELECT
  order_id,
  item.sku,
  item.qty,
  item.net_amount
FROM `analytics_core.fact_orders`,
UNNEST(line_items) AS item
WHERE order_date = DATE '2026-04-10';
```

Regel:

- `UNNEST` ist kein Sonderfall, sondern Kerntechnik für analytische Modelle mit verschachtelten Feldern.

4.6. Window Functions für BI- und Analytics-Muster

BigQuery ist stark bei:

- Running Totals
- Rankings
- Sessionisierung
- Vorher/Nachher-Vergleichen
- Anreicherungen ohne Self-Join-Overkill

Beispiele:

- `ROW_NUMBER()`
- `SUM() OVER (...)`
- `LAG()` / `LEAD()`
- `QUALIFY` Gerade `QUALIFY` hilft oft, BI-nahe Patterns lesbarer zu halten.

4.7. Inkrementelle Loads mit `MERGE`

In der Praxis ist `MERGE` eines der wichtigsten DML-Werkzeuge:

```
MERGE `analytics_core.dim_customer` AS target
USING `analytics_stage.customer_delta` AS src
ON target.customer_id = src.customer_id
WHEN MATCHED THEN
  UPDATE SET
    customer_name = src.customer_name,
    segment = src.segment,
    updated_at = CURRENT_TIMESTAMP()
WHEN NOT MATCHED THEN
  INSERT (customer_id, customer_name, segment, updated_at)
  VALUES (src.customer_id, src.customer_name, src.segment, CURRENT_TIMESTAMP());
```

Praxisregel:

- `MERGE` ist maechtig, aber bei sehr grossen Tabellen operativ relevant für Kosten, Locks im fachlichen Sinne und Change-Risiko.
- Vor grossen `MERGE`-Wellen helfen Snapshots, partitionierte Verarbeitung und klare Deduplizierungslogik.

4.8. Multi-Statement, Skripte, Transaktionen und Prozeduren

BigQuery kann mehr als einzelne Queries. Verfügbare Muster:

- Multi-Statement-Skripte
- Stored Procedures
- Transaktionen für Datenmodifikationen
- Fehlerbehandlung und deklarative Variablen im Skriptkontext

Das ist besonders wertvoll für:

- kontrollierte Batch-Prozesse,
- ELT-Schritte,
- fachlich gekapselte Lade- und Publikationslogik.

4.9. Views, Table Functions und UDFs

Diese Werkzeuge lösen unterschiedliche Probleme:

- **Views**: logische Abstraktion und Konsumvereinfachung
- **Materialized Views**: Performance für passende Muster
- **Table Functions**: parametrische, wiederverwendbare Tabellenlogik
- **UDFs**: wiederverwendbare Berechnungen

Praxisregel:

- Nutze Views für Konsum und Semantik.
- Nutze UDFs sparsam und nur bei echter Wiederverwendung.
- Vermeide, dass geschäftskritische Kernlogik in dutzenden Einzelfiles dupliziert wird.

4.10. Search, Vector, GIS, ML - Spezialwerkzeuge mit Mehrwert

BigQuery bietet über klassisches SQL hinaus moderne Funktionen:

- Search Indexes für schnelle Suchpfade in grossen Text-/JSON-dominierten Tabellen.
- Vector Search und Vector Indexes für Embedding-nahe Suchfaelle.
- Geospatial-Funktionen für Geo-Analysen.
- BigQuery ML für modellnahe Analysen in SQL.

Wichtig:

- Diese Features sind wertvoll, aber sie ersetzen keine saubere Tabellen- und Workload-Architektur.
- Einige Features sind editions- oder kostenrelevant und sollten bewusst eingefuehrt werden.

4.11. So setzt du es um - Checkliste

- Standardisiere GoogleSQL als Team-Default.
- Erzwinge bei Review von grossen Queries Projektions- und Partition-Disziplin.
- Nutze `MERGE`, Views und Routinen nur mit klarer Betriebsbegruendung.
- Halte wiederverwendbare Logik zentral statt in BI-Tools zu duplizieren.
- Führe Spezialfeatures wie Search oder Vector erst mit klarer Use-Case-Begruendung ein.

4.12. SQL-Stil für Produktionsreife

Bewaehrte Stilregeln:

- Fachliche CTE-Namen statt `t1`, `t2`, `tmp`.
- Filter früh im Plan, besonders für Partitionen.
- Spaltenlisten explizit statt blindes `SELECT *`.
- Kommentare nur dort, wo fachliche Absicht sonst unklar bleibt.
- DDL, DML und analytische Queries als versionierte Artefakte behandeln.

4.13. Beispiel: Query-Review-Fragen

Vor Freigabe einer produktiven Query sollten mindestens beantwortet sein:

- Nutzt die Query Partition Pruning?
- Liest sie nur benötigte Spalten?
- Ist ein Join fachlich und technisch notwendig?
- Würde eine Materialisierung den Dauerbetrieb stabilisieren?
- Ist die Security-Wirkung über Views, Policy Tags oder Dataset-Grenzen verstanden?

---

5. Datenaufnahme, Transformation & Delivery

5.1. Batch-Loads für robuste Grundversorgung

Batch-Loads sind oft die stabilste und guenstigste Basis für viele Datenstroeme. Typische Muster:

- Parquet/Avro aus Cloud Storage laden.
- Rohdaten in `raw`-Tabellen materialisieren.
- Validierung und Harmonisierung im Anschluss.

Vorteile:

- Gute Reproduzierbarkeit.
- Klare Ladefenster.
- Saubere Fehlerbehandlung und Replay-Pfade.

5.2. Streaming mit der Storage Write API

Für kontinuierliche Datenstroeme ist die Storage Write API der zentrale Baustein. Wichtige Denkmuster:

- Default Stream für at-least-once Semantik.
- Commit-/Offset-basierte Muster für exactly-once-nahe Anforderungen.
- Streaming ist nicht nur Technik, sondern auch Kosten- und Betriebsentscheidung.

Praxisregel:

- Nutze Streaming dort, wo echte Frischeanforderung besteht.
- Alles, was auch im 5-, 15- oder 60-Minuten-Takt reichen würde, sollte bewusst gegen Batch abgewogen werden.

5.3. CDC und inkrementelle Verarbeitung

Viele produktive BigQuery-Setups leben von inkrementellen Änderungen:

- Quellsystem exportiert Deltas.
- Deltas landen in Stage-Tabellen.
- `MERGE` oder deduplizierende CTAS-Patterns aktualisieren Kernobjekte.

Wichtig:

- Idempotenz vor Geschwindigkeit.
- Jede Delta-Logik braucht Schlüssel, Reihenfolge und Konfliktregel.

5.4. Orchestrierung: Scheduled Queries, SQL-Pipelines und Build-Ketten

Transformation in BigQuery ist nicht nur Query-Ausfuehrung, sondern Lieferkette. Typische Optionen:

- Scheduled Queries für einfache, klare Takte.
- SQL-zentrierte Build-Tools wie Dataform oder dbt für versionierte Modelle.
- Workflow-Orchestratoren für komplexere Abhängigkeiten und externe Schritte.

Heuristik:

- So simpel wie möglich, so explizit wie nötig.

5.5. Data Contracts und Datenqualität

Eine produktive Plattform braucht Mindestgarantien:

- Pflichtfelder.
- Schluesselregeln.
- Akzeptierte Wertebereiche.
- Freshness und Ladevollstaendigkeit.

Ohne diese Regeln entstehen später:

- Stille KPI-Abweichungen.
- schwierige Debugging-Situationen.
- politische Diskussionen statt technischer Ursachenanalyse.

5.6. Backfills und Replay als Kernfaehigkeit

Backfills sind kein Sonderfall, sondern normaler Teil des Plattformlebens:

- Quellsystem liefert nachträglich korrigierte Daten.
- Logik wird angepasst.
- Historische Kennzahlen müssen neu berechnet werden.

Deshalb braucht jede Pipeline Antworten auf:

- Wie weit darf ich neu rechnen?
- Wie erkenne ich betroffene Partitionen?
- Wie isoliere ich den Backfill von interaktiver Last?
- Wie prüfe ich das Ergebnis?

5.7. Delivery in Richtung BI, APIs und Datenprodukte

Delivery meint:

- Dashboards.
- Fachdatensaetze.
- Exporte.
- Sharing über Views oder Data Products.

Wichtige Regel:

- Gib Konsumenten möglichst stabile, klar dokumentierte Objekte.
- Lass Roh- oder Core-Tabellen nicht blind direkt an jede BI-Schicht.

5.8. Mini-Projekt: Pipeline-Skizze

Für die Commerce-Plattform:

1. Web- und App-Events kommen near real time über die Storage Write API.
2. ERP, CRM und Ads werden täglich oder mehrmals täglich batchweise geladen.
3. In `raw` bleiben Quellen nachvollziehbar.
4. `core` harmonisiert Kunden, Produkte, Kampagnen und Orders.
5. `mart` liefert Dashboards für Marketing, Sales, Finance und Operations.
6. Backfills laufen in separaten Fenstern und konkurrieren nicht mit Dashboards.

5.9. So setzt du es um - Checkliste

- Entscheide je Datenquelle bewusst zwischen Batch und Streaming.
- Definiere Idempotenz- und Replay-Strategien für jede Pipeline.
- Trenne Stage-/Delta-Tabellen von konsumierbaren Kernobjekten.
- Versioniere Transformationslogik statt SQL lose im UI zu sammeln.
- Führe Data-Quality-Checks und Freshness-Metriken ein.
- Trenne Backfills von interaktiver Produktivlast.

5.10. Workload-Trennung für Pipelines

Ein häufig ueberschaetztes Problem ist nicht die einzelne Query, sondern Konkurrenz:

- Dashboards laufen morgens.
- Notebook-User explorieren parallel.
- Backfills werden spontan angestossen.
- Geplante Transforms starten zur vollen Stunde. Wenn das alles denselben Rechenpool trifft, entsteht Instabilitaet. Deshalb:
- Hintergrundlast logisch oder technisch trennen.
- Kritische Delivery-Pfade priorisieren.
- Kosten und Latenzen pro Workload sichtbar machen.

5.11. Quoten und Betriebsgrenzen mitdenken

Load Jobs, Tabellenmodifikationen und API-Pfade haben Grenzen. Das ist relevant für:

- viele kleine Loads,

- aggressive Micro-Batch-Strategien,
- Copy-/Overwrite-intensive Architekturen.

Praxisregel:

- Nicht nur die SQL-Logik, sondern auch das Betriebsprofil der Pipeline designen.

---

6. Betrieb, Beobachtbarkeit & Troubleshooting

6.1. Beobachtbarkeit beginnt bei Jobs, nicht erst bei Dashboards

Die wichtigste operative Sicht in BigQuery ist oft die Job-Sicht. Beobachte mindestens:

- Laufzeit
- `total_bytes_processed`
- `total_slot_ms`
- Fehlercodes
- `statement_type`
- Nutzer, Projekt, Region und Quelle Das hilft dir, Performance-, Kosten- und Governance-Themen zusammenzulesen.

6.2. INFORMATION_SCHEMA als Betriebswerkzeug

`INFORMATION_SCHEMA` ist in BigQuery kein Nice-to-have, sondern Kernwerkzeug. Typische Einsaetze:

- Query-Historie auswerten
- teure Jobs finden
- Slot-Verbrauch sichtbar machen
- Index-, Reservation- oder Tabellenmetadaten prüfen

Beispiel:

```
SELECT
  creation_time,
  user_email,
  statement_type,
  total_bytes_processed,
  total_slot_ms
FROM `region-REGION_NAME`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE creation_time >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY)
  AND job_type = 'QUERY'
ORDER BY total_bytes_processed DESC
LIMIT 50;
```

6.3. Execution Graph und Performance Insights lesen

BigQuery liefert für Queries einen Ausfuehrungsgraphen und Performance-Hinweise. Wichtige Signale:

- Slot Contention
- Shuffle-Engpaesse
- Join-Skew
- BI-Engine-Nichtnutzung
- starke Input-Differenzen bei Regressionsfaellen

Das Ziel ist nicht nur "Query war langsam", sondern:

- Welche Stage war teuer?
- War das Problem Datenmenge, Konkurrenz oder Query-Struktur?

6.4. Häufige Fehlerbilder

Typische Fehler im Alltag:

- Fehlende Partition-Filter auf grossen Tabellen.
- Ueberraschend hohe Bytes durch `SELECT *`.
- Join-Skew oder unpassende Join-Reihenfolge.
- Konkurrenz zwischen Background und Interactive Workloads.
- Schema Drift aus Quellen.
- Permissions- oder Location-Fehler.
- `quotaExceeded` bei Load- oder Table-Operationen.

6.5. Troubleshooting-Muster für Kostenexplosionen

Wenn Kosten plötzlich steigen, prüfe zuerst:

1. Welche Jobs oder Nutzer sind neu teuer?
2. Wurde ein neues Dashboard, Notebook oder Export ausgerollt?
3. Fehlt Pruning oder Projektion?
4. Wurde von nativen Tabellen auf externe Dateien umgeschwenkt?
5. Gibt es häufige Overwrites, die Storage-/Time-Travel-Folgen haben? Kostenfehler sind oft kein Abrechnungsproblem, sondern ein Architektur- oder Freigabeproblem.

6.6. Troubleshooting-Muster für Latenzprobleme

Wenn Queries langsamer werden:

- Vergleiche `bytes processed` und `slot ms` historisch.
- Prüfe Konkurrenz im Zeitfenster.
- Prüfe, ob Materialized Views oder BI Engine greifen.
- Suche Join- oder Shuffle-Hotspots.
- Vergleiche neue Releases von SQL, Semantik-Layer oder BI-Modellen.

6.7. Restore, Rueckfall und kontrollierte Fehlerbehebung

Für produktive Datenplattformen gilt:

- Keine grossen `MERGE`- oder Rewrite-Aktionen ohne Rueckfallpfad.
- Snapshots vor kritischen Umbauten.
- Time Travel für schnelle Ersthilfe.
- Clones für Vergleich und Validierung. So bleibt Fehlerbehebung reproduzierbar und nicht improvisiert.

6.8. So setzt du es um - Checkliste

- Definiere ein Kernset aus Job-, Kosten- und Freshness-Metriken.
- Führe tägliche oder woechentliche Reviews teurer Queries ein.
- Nutze `INFORMATION_SCHEMA` systematisch, nicht nur bei Incidents.

- Lege Runbooks für Kosten-, Latenz- und Berechtigungsprobleme an.
- Teste Restore-Wege mit Snapshot, Clone oder Time Travel.
- Halte bekannte Limits und Quoten pro Pipeline dokumentiert.

6.9. Runbook-Vorlage pro Kerndatenprodukt

Eine einfache, aber wirksame Vorlage:

- Datenprodukt / Tabelle / View
- Geschäftszweck
- Frische-SLO und Verfuegbarkeitsanspruch
- Kritische Upstream-Quellen
- Kritische Queries / Dashboards
- Normale Kosten- und Laufzeitbereiche
- Bekannte Fehlerbilder
- Erste 5 Diagnoseabfragen
- Eskalationskette

6.10. Post-Incident-Lernen in BigQuery-Teams

Nach jedem Incident sollte kurz beantwortet werden:

1. Was war das erste messbare Signal?
2. War es Kosten-, Frische-, Latenz- oder Security-getrieben?
3. Welche Modell- oder Architekturannahme war falsch?
4. Welche Leitplanke hätte früher geholfen?
5. Welche Runbook- oder Ownership-Aenderung folgt daraus? Der Fokus liegt auf Systemverbesserung, nicht auf Schuldzuweisung.

6.11. Betriebsmetriken für den Alltag

Sinnvolle Standardmetriken:

- Frische kritischer Tabellen
- Laufzeit kritischer Pipelines
- Top-Queries nach Bytes und Slot-Zeit
- Anzahl fehlgeschlagener Jobs
- Slot-Contention-Hinweise
- Storage-Wachstum nach Dataset
- Anzahl sicherheitsrelevanter Zugriffsobjekte wie Authorized Views oder Policy Tags

---

7. Performance, Skalierung & Kostenkontrolle

7.1. Reduziere zuerst gelesene Daten, nicht nur CPU-Zeit

Der wichtigste Hebel ist meist:

- weniger Bytes lesen
- weniger irrelevante Spalten lesen

- weniger breite Zwischenmengen erzeugen

Praxisregeln:

- Kein `SELECT *` auf grossen Tabellen.
- Filter auf Partitionen früh setzen.
- Cluster-Spalten in realen Filterpfaden nutzen.
- Zwischenresultate materialisieren, wenn sie mehrfach verwendet werden.

7.2. Partitionierung und Clustering als Dauerhebel

Diese beiden Entscheidungen wirken nicht einmalig, sondern bei jeder Abfrage. Deshalb gilt:

- Jede grosse Tabelle ohne klare Partitionierungsbegruendung ist ein Risiko.
- Jede grosse Tabelle mit unklarer Cluster-Wahl ist ein Optimierungskandidat.

Nutze auch Empfehlungen und Beobachtbarkeit:

- Partition-/Cluster-Recommender können Hinweise geben.
- Dennoch bleibt die Fach- und Workload-Pruefung deine Verantwortung.

7.3. Materialized Views, BI Engine und gezielte Voraggregation

Nicht jede Beschleunigung gehört in dieselbe Kategorie:

- **Materialized Views** helfen bei passenden SQL-Mustern.
- **BI Engine** hilft vor allem für wiederkehrende, interaktive BI-Abfragen.
- **Aggregations-/Serving-Tabellen** helfen bei strikten SLOs und klaren Konsummustern.

Die richtige Frage ist:

- Muss die Abfrage flexibel sein?
- Muss sie billig sein?
- Muss sie sehr schnell sein? Oft kannst du nicht alle drei Ziele maximal gleichzeitig haben.

7.4. Workload-Management mit Reservations

Ab einer gewissen Reife ist Kostenkontrolle ohne Workload-Steuerung schwer. Reservations helfen bei:

- Vorhersagbarkeit.
- Isolation.
- SLO-Schutz.
- planbarerem FinOps.

Typische Trennung:

- Interactive BI
- ETL / Background
- Data Science / Notebooks
- Dev/Test / Sandbox

7.5. Storage-Kosten bewusst steuern

Wichtige Hebel:

- Long-Term Storage nutzen, wenn Daten selten verändert werden.
- Partition Expiration für kurzlebige Detaildaten.
- Archivierung nach Cloud Storage für Altbestaende.
- Unter physischer Storage-Abrechnung Overwrite-Muster kritisch betrachten, weil Time Travel und Fail-Safe Nachwirkungen haben können.

7.6. Externe Tabellen und Kostenrealismus

Externe Datenzugriffe können elegant sein, aber sie sind nicht automatisch guenstig oder schnell. Prüfe bewusst:

- Metadaten-Overhead
- Dateianzahl und Dateigroessen
- Pruning-Faehigkeit
- ob native Materialisierung für Kernpfade sinnvoller wäre Gerade bei sehr vielen kleinen Dateien oder hochfrequentem Zugriff kippt die Bilanz schnell.

7.7. Typische Kostentreiber

- `SELECT *` in BI- oder Notebook-Abfragen
- Fehlendes Partition Pruning
- zu viele nahezu identische Reports mit leicht variierender SQL
- unkontrollierte Sandboxes
- teure `MERGE`- oder Rewrite-Muster
- Konkurrenz, die in Reservation-Umgebungen mehr Slots zieht als geplant
- unnötige Recomputes statt stabiler Serving-Objekte

7.8. Mini-Projekt: Tuning-Loop

1. Identifiziere die 20 teuersten Abfragen der letzten 14 Tage.
2. Mappe sie auf Tabellen, Teams und Dashboards.
3. Prüfe Projektions- und Partition-Disziplin.
4. Validiere Cluster-Spalten und Join-Muster.
5. Entscheide, ob Materialisierung oder Reservation-Trennung sinnvoller ist.
6. Miss erneut Kosten, Laufzeit und Frische.

7.9. So setzt du es um - Checkliste

- Top-Queries nach Bytes und Slot-Zeit sind regelmäßig transparent.
- Grosse Fakttabellen erzwingen oder erwarten saubere Partition-Filter.
- Serving-Objekte für wiederkehrende BI-Pfade sind bewusst definiert.
- Interactive und Background Workloads sind getrennt oder klar priorisiert.
- Storage-Lebenszyklen sind dokumentiert.
- Kostenanomalien lösen einen klaren Diagnosepfad aus.

7.10. Release-Gates für Performance und FinOps

Ein unterschätzter Ansatz:

- Neue Dashboards, größere SQL-Modelle oder neue Self-Service-Datasets gehen nur live, wenn ein kleines Gate erfüllt ist.

Beispiel-Gates:

- Kein Vollscan auf definierter Fakttabelle ohne Sonderfreigabe.
- Query-Kosten unter vereinbartem Schwellwert.
- Partition-Pruning nachgewiesen.
- Dashboard-Latenz unter Zielwert.
- Security- und Ownership-Check abgeschlossen.

7.11. Kostenkontrolle durch Datenklassifizierung

Ein praktikables Modell:

- **Tier A**: geschaeftskritische Datenprodukte, hohe Aktualitaet, strenge Kostenbeobachtung.
- **Tier B**: wichtige operative und analytische Standarddaten.
- **Tier C**: explorative, kurzlebige oder experimentelle Daten.

Diese Klassifizierung hilft bei:

- Reservations-Zuteilung,
- Storage-Lebenszyklus,
- Quality Gates,
- Security-Intensitaet,
- FinOps-Berichterstattung.

7.12. Benchmarking, das wirklich hilft

Ein sinnvoller BigQuery-Benchmark misst nicht nur Mittelwerte. Wichtig sind:

- Kosten pro Abfrageklasse
- Laufzeit bei Konkurrenz
- Verhalten über reale Zeitfenster
- Frische unter Pipeline-Last
- Wirkung von Materialisierung, Partitionierung und Reservation-Trennung Synthetische Einzelfall-SQL ohne echten Nutzerkontext führt oft zu falschen Schluessen.

---

8. Sicherheit, Governance & Compliance

8.1. Sicherheitsgrundsatz

Eine funktionierende BigQuery-Umgebung ist noch keine sichere BigQuery-Umgebung. Produktionsreife braucht mindestens:

- klares IAM-Modell,
- dataset- und objektbezogene Ownership,
- Schutz sensibler Spalten und Zeilen,
- saubere Auditierbarkeit,
- kontrollierte Freigabepfade.

8.2. IAM: Least Privilege zuerst

BigQuery nutzt IAM als Grundmodell für Zugriff. Wichtige Praxisregeln:

- Keine breit gestreuten Admin-Rollen für den Alltag.
- Service Accounts pro Plattformfunktion statt Sammelkonten.
- Rollen möglichst nah am benötigten Ressourcenumfang vergeben.
- Projektweite Rechte nur dort, wo sie wirklich nötig sind.

8.3. Dataset-, Tabellen- und View-Grenzen bewusst nutzen

Access-Kontrolle ist nicht nur Rollenvergabe, sondern Ressourcenarchitektur. Hilfreiche Muster:

- Datasets als Sicherheits- und Ownership-Grenzen.
- Tabellen-/View-Rechte für feingranulare Freigaben.
- Separate Konsumobjekte für Fachbereiche statt Rohdatenzugriff.

8.4. Row-Level Security, Column-Level Security und Authorized Views

BigQuery bietet mehrere Ebenen für feingranularen Zugriff:

- **Row-Level Security** für zeilenbezogene Filter.
- **Column-Level Security** über Policy Tags.
- **Authorized Views** für freigegebene Sichten.
- **Data Masking** über Data Policies für abgestufte Sichtbarkeit.

Wichtig:

- Diese Mechanismen sind kombinierbar, aber nicht kostenlos im Betriebs- und Performance-Sinne.
- Row-Level Security beteiligt sich nicht an Partition-Pruning und kann Beschleunigungsmechanismen beeinflussen.

8.5. Policy Tags und Masking als Governance-Werkzeuge

Policy Tags helfen, sensible Spalten systematisch zu klassifizieren. Data Masking hilft, Nutzern nur die benötigte Sichtbarkeit zu geben. Praxisregel:

- Nicht jede Sicherheit mit Views modellieren.
- Nicht jede Sicherheit mit Policy Tags modellieren.
- Waehle das Muster, das fachlich, technisch und auditierbar passt.

8.6. Verschluesselung, Perimeter und editionssensitive Features

BigQuery verschluesselt Daten standardmaessig at rest und in transit. Je nach Anforderung kommen weitere Mechanismen hinzu:

- CMEK für ausgewählte Schluesselkontrolle
- VPC Service Controls für Perimeter-Schutz
- organisationsweite Richtlinien für Export-, Sharing- und Zugriffspfade

Wichtig:

- Einige Features sind editions- oder umgebungsabhaengig.

- Security-Design sollte daher immer gegen die aktuelle Editions-Matrix und Regionalstrategie geprüft werden.

8.7. Audit, Lineage und Nachvollziehbarkeit

Governance wird erst belastbar, wenn du sie nachweisen kannst. Dazu gehören:

- Audit Logs
- dokumentierte Freigaben
- nachvollziehbare Objekt-Ownership
- Sichtbarkeit über Datenfluesse, Ableitungen und sensiblen Konsum

Die Leitfrage ist:

- Wer konnte was wann sehen, ändern oder verarbeiten?

8.8. So setzt du es um - Checkliste

- IAM folgt nachweisbar Least Privilege.
- Datasets sind Sicherheits- und Ownership-Grenzen, nicht nur Ordnerersatz.
- Sensible Spalten sind klassifiziert und über Policy Tags oder Masking abgedeckt.
- Row-Level Security wird nur dort eingesetzt, wo sie fachlich sinnvoll ist.
- Freigaben für Fachbereiche laufen über definierte Konsumobjekte.
- Audit- und Review-Prozesse sind etabliert.

8.9. Governance-Metriken für Reviews

Sinnvolle Kennzahlen:

- Anteil sensibler Spalten mit dokumentierter Klassifikation
- Anzahl ueberprivilegierter Service Accounts
- Anzahl nicht dokumentierter Authorized Views
- Anteil produktiver Datasets mit benanntem Owner
- Anzahl ausstehender Rollenrezertifizierungen
- Anzahl Tabellen mit PII ohne definierte Schutzmassnahme

8.10. Security-Drills statt Dokumentationsillusion

Beispielszenarien:

- Falsche Freigabe auf einem Mart-Dataset
- Unmaskierter Zugriff auf sensible Spalten
- Datenexport in nicht erlaubte Region
- Service Account mit zu breiten Rechten

Ziel:

- Erkennungsgeschwindigkeit messen
- Freigabeketten testen
- technische und organisatorische Reaktion prüfen

8.11. Least Privilege in der Praxis erzwingen

Wirksame Mechanismen:

- Rollen und Grants über IaC oder versionierte Plattformprozesse verwalten.
- Ausnahmerechte mit Ablaufdatum versehen.
- Rezertifizierungen quartalsweise durchfuehren.
- Neue Datasets nur mit Owner, Klassifikation und Mindestleitplanken freigeben.

8.12. Governance als Produktivfaehigkeit verstehen

Governance ist nicht "Bremse für Analytics", sondern Voraussetzung für sichere Skalierung. Ohne Governance steigt mit jedem neuen Team:

- das Kostenrisiko,
- das Sicherheitsrisiko,
- die semantische Inkonsistenz,
- der Incident-Aufwand.

Mit guter Governance steigt hingegen:

- Wiederverwendbarkeit,
- Vertrauen in Zahlen,
- Freigabetempo bei geringerer Unsicherheit,
- langfristige Plattformqualitaet.

---

9. Weiterführende Links + Übungsaufgaben

9.1. Offizielle Kernquellen (Google Cloud)

Produktueberblick:

- BigQuery Overview
- BigQuery Dokumentation

Compute & Kosten:

- BigQuery Editions
- Reservations / Workload Management
- Kosten-Best-Practices

Modellierung & Performance:

- Partitioned Tables
- Clustered Tables
- Query Performance Best Practices
- Nested and Repeated Fields
- Storage Best Practices

SQL & Datenverarbeitung:

- GoogleSQL Intro

- Storage Write API
- Materialized Views

Betrieb & Troubleshooting:

- Query Insights
- JOBS INFORMATION_SCHEMA
- Time Travel
- Table Clones
- Table Snapshots

Sicherheit & Governance:

- Access Control Intro
- Row-Level Security
- Column-Level Security
- Data Masking
- Authorized Views

Spezialfunktionen:

- Search Indexes
- Vector Search

9.2. Übungsaufgaben (steigende Schwierigkeit)

1. **Tabellendesign-Basis (leicht)**

- Modelliere eine partitionierte und geclusterte `fact_orders`-Tabelle.
- Begruende Partitionierung, Clustering und `require_partition_filter`.

2. **Query-Kosten senken (leicht bis mittel)**

- Nimm drei teure Beispielabfragen.
- Reduziere gescannte Bytes durch Projektion, Partition Pruning und bessere Filter.

3. **Inkrementeller Load (mittel)**

- Lade Kundendeltas in eine Stage-Tabelle.
- Aktualisiere die Zieltabelle per `MERGE`.
- Dokumentiere Idempotenz und Rueckfallstrategie.

4. **Streaming-Pfad (mittel)**

- Definiere ein Eventing-Pattern mit Storage Write API.
- Lege fest, wo at-least-once genügt und wo Deduplizierung nötig ist.

5. **Workload-Isolation (mittel bis schwer)**

- Trenne BI- und Background-Workloads logisch oder rechnerisch.
- Begruende die Reservations-/Assignments-Strategie.

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

- Entwirf ein Modell mit Dataset-Grenzen, Policy Tags und Authorized Views.

- Definiere, welche Rollen Rohdaten, maskierte Daten und aggregierte Marts sehen dürfen.

7. **Cost & Incident Drill (schwer)**

- Simuliere plötzlich gestiegene Query-Kosten.
- Führe Diagnose über `INFORMATION_SCHEMA`, Query Insights und Runbook durch.
- Leite dauerhafte Guardrails ab.

9.3. Abnahmekriterien für "produktionsreif"

- Kernabfragen haben nachgewiesene Kosten- und Latenzprofile.
- Grosse Tabellen sind sinnvoll partitioniert und bei Bedarf geclustert.
- Ingestion-, Replay- und Backfill-Pfade sind reproduzierbar.
- Kritische Workloads sind rechnerisch oder organisatorisch getrennt.
- Sensible Daten sind über IAM, Policy Tags, Masking oder Views kontrolliert.
- Restore- und Incident-Prozesse sind getestet.

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

Pro-Level heisst nicht, jede Funktion auswendig zu kennen. Pro-Level heisst:

- Du modellierst für Fragen, Kosten und Governance gleichzeitig.
- Du steuerst Compute bewusst statt nur auf Query-Texte zu schauen.
- Du baust Delivery-Pfade mit klaren SLOs und Ownership.
- Du kannst Kosten- und Latenzprobleme über Jobs, Plaene und Workloads reproduzierbar analysieren.
- Du behandelst Security und Datenfreigabe als Architekturthema, nicht als Nacharbeit. Wenn diese Punkte erfüllt sind, ist BigQuery nicht nur ein bequemes SQL-Tool, sondern eine belastbare Datenplattform.

9.5. So setzt du es um - Checkliste

- Alle offiziellen Kernquellen sind im Team-Wiki als Primärreferenz verankert.
- Uebungen 1 bis 7 sind mit messbaren Ergebnissen dokumentiert.
- Für zentrale Tabellen existieren Modell-, Kosten- und Security-Dokumentation.
- Plattform-, Fach- und Security-Owner sind benannt.
- Ein wiederholbarer 90-Tage-Verbesserungsplan ist vereinbart.

9.6. 90-Tage-Umsetzungsfahrplan

Tage 1-30

- Kernfragen, Workloads und Dataset-Grenzen definieren.
- Tabellenmodell für Raw/Core/Mart festziehen.
- Erste Kosten- und Security-Guardrails einziehen. **Tage 31-60**
- Inkrementelle Pipelines, Backfills und Runbooks stabilisieren.
- Top-Queries optimieren.
- Reservations-/Workload-Modell für Produktivpfade schaerfen. **Tage 61-90**
- Security- und Governance-Drills fahren.
- FinOps-Review mit realen Verbrauchsmustern etablieren.

- Restore-, Snapshot- und Incident-Prozesse nachweisbar ueben. Nach 90 Tagen sollte das Team nicht nur Queries schreiben können, sondern Plattformverhalten bei Kosten, Latenz und Zugriff kontrolliert steuern.

9.7. Self-Assessment für Teams

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

- Wir kennen unsere teuersten Queries und können sie erklären.
- Unsere grossen Tabellen sind bewusst partitioniert und begründet geclustert.
- Wir haben für kritische Datenprodukte klare Owner.
- Wir können Backfills reproduzierbar und ohne Chaos ausführen.
- Unsere interaktiven und Hintergrund-Workloads konkurrieren nicht unkontrolliert.
- Wir haben ein belastbares Modell für sensible Spalten und Freigaben.
- Wir können einen Kosten- oder Berechtigungsincident strukturiert behandeln.

Auswertung:

- 0-5 Punkte: frühe Pionierphase, hohe Betriebs- und Governance-Risiken.
- 6-10 Punkte: solide Basis, aber inkonsistent.
- 11-14 Punkte: gutes Produktionsniveau.

9.8. Nächste Ausbaustufen

Wenn die Basis stabil steht, sind sinnvolle nächste Schritte:

- Performance- und Kosten-Gates im Releaseprozess verankern.
- Query- und Cost-Reviews als festen Teamprozess etablieren.
- Datenklassifizierung tiefer mit Masking, Policy Tags und Produkt-Ownership verbinden.
- Search-, Vector- oder ML-Funktionen nur dort einfuehren, wo das Basismodell bereits sauber ist. Ergaenze dazu einen quartalsweisen "BigQuery Platform Review", der nicht nur Kostenberichte betrachtet, sondern bewusst die Verbindung zwischen Produktwirkung und Plattformzustand herstellt: Welche Dashboards und Datenprodukte haben die größte Lastveraenderung erzeugt, welche Tabellen verlieren ihre Partitionierungs- oder Clustering-Wirkung, wo führen neue Teams zu Berechtigungsdruck, wo entstehen neue Marts statt Wiederverwendung, und welche Workloads brauchen inzwischen eine saubere Reservation-Trennung statt opportunistischer Mitnutzung? Dieser Review sollte immer mit konkreten Folgeentscheidungen enden, etwa neue Guardrails für teure Fakttabellen, Umbau eines konsumkritischen Reports auf Aggregationstabellen, Schliessung eines zu breiten Dataset-Zugriffs, Vereinheitlichung von Kernlogik oder gezielte Archivierung alter Detaildaten. Auf diese Weise bleibt BigQuery keine Ansammlung einzelner Queries, sondern eine aktiv geführte Analyseplattform, deren Kosten, Geschwindigkeit, Sicherheitsniveau und Nutzbarkeit über Jahre bewusst gesteuert werden.

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 11.04.2026 wieder; maßgeblich ist die Online-Fassung unter https://www.thinkbigger.at/wissen/bigquery/