Power BI on Databricks Best Practices Cheat Sheet

This document summarizes the most relevant best practices when using Power BI

PHASE	DATA PREPARATION	SQL SERVING	POWER BI INTEGRATION	POWER BI REPORT DESIGN
Read if you are	Preparing the dataset that will be served for the report	Setting up and configuring the Databricks SQL Warehouse to which Power BI connects	Configuring the integration between Power BI and Databricks	Designing semantic models and reports in Power BI
Your objective	▪ **Implement an efficient data model** ▪ **Optimize storage layer for query performance**	▪ **Meet performance SLA and scalability** ▪ **Keep costs under control**	▪ **Optimize semantic model for performance** ▪ **Leverage unified governance**	▪ **Create fast and efficient reports and dashboards** ▪ **Inherit the optimizations performed in all the other layers**
Best practices: What you should do	▪ Adopt **medallion architecture** on **Delta Lake** (link) **and serve the Gold layer only** (link) → leverage benefits of Delta and aggregated data for better report performance. ▪ When designing your data model, opt if possible for a **star schema** (link, link) → Power BI performs better. ▪ Leverage **SQL views** and **persisted tables** with the required granularity (e.g., date/time) → improve performance by pre-aggregating data for the most common and resource-intensive queries. ▪ Declare **primary and foreign keys** (link), and use **RELY** (link) where possible → Power BI can leverage this information to automatically create table relationships, and the Databricks SQL engine can optimize queries using PK constraints. ▪ **Avoid** using **wide data types** and **high-cardinality** columns → reduce Power BI semantic model size and improve query performance. ▪ Use **auto-generated columns** when you need to generate a derived value from other existing columns (link) → persisted columns minimize the need for calculating values at query time. ▪ Use **Liquid Clustering** (link); alternatively, use Z-ordering (link) and consider partitioning larger tables >1TB (link) → improve query performance by efficient file and data skipping. ▪ **Optimize your data layout** by using predictive optimization (link) or running VACUUM (link) and OPTIMIZE (link) → improve performance by deleting old files and optimizing the physical data layout. ▪ Periodically **compute statistics** (link) or use **automatic statistics** (currently in preview, link) → improve performance by choosing a more efficient join strategy. ▪ Evaluate using **materialized views** (link) to calculate results incrementally using the latest data in the source tables → improve performance by leveraging precomputed results.	▪ Always use a **SQL warehouse** (and not all-purpose clusters) (link) → SQL warehouses are optimized for BI workloads. ▪ Use SQL Serverless warehouse (link) → optimal price/performance and delivers instant and elastic compute. Leverage Serverless query result cache even when SQL Warehouse is scaled or restarted. ▪ Enable SQL warehouse **Auto stop** (scale-to-zero) only if the SLA permits; the SQL warehouse will take just 5–10s to start up when the first query arrives → save costs when the warehouse is not used. ▪ Use **higher cluster size for larger datasets** (link) → the larger the cluster (M, L, XL, etc.), the faster complex queries run. If having only simple, short-running queries, don't increase the size (might be slower due to data shuffling). ▪ Use **SQL Warehouse scaling** (min/max cluster count) (link) → **handle more concurrent users / queries**. SQL Warehouse scales out to handle the increased workload. When hitting limits, queries get queued, not rejected. ▪ If expecting many concurrent queries, **increase the minimum number of clusters** → prevent queueing queries waiting for scaling out. ▪ Use the **same SQL Warehouse** whenever the same dataset is queried → will leverage the various layers of caching available, increasing performance. ▪ Use **separate SQL warehouses** for different workloads and/or business units → right-size SQL warehouses to achieve better performance and reasonable costs. ▪ If unsure about sizing, start with a **Medium** SQL Warehouse scaling between 1 and 10; **monitor query response time and scaling** → then adjust the sizing based on the observed results. ▪ **Leverage** the Query History (link) and SQL warehouse events (link) **system tables** to programmatically monitor SQL Warehouses and query performance → allows for identifying performance bottlenecks and issues, implementing more detailed analysis and setting up alerts.	▪ Ensure that Power BI and Databricks are hosted as closely as possible, ideally in **the same region** → this would minimize network latency and may help avoid cross-region traffic costs. ▪ Use the most appropriate Power BI storage mode: **Direct Query** for Fact tables, **Dual** for Dimension tables (NOT Import) (link) → let Power BI generate more efficient SQL queries. ▪ Evaluate where and how to use **composite models** (link) → allows a mixed usage of DirectQuery, Dual, Import mode tables, as well as Aggregation and Hybrid tables. ▪ Use **hybrid tables** whenever you need aggregated historical data augmented with detailed real-time data for the same table (link) → efficient and quick in-memory queries combined with the latest data changes directly from the source. ▪ In Import mode, use **table partitioning** (link). Alternatively use **incremental refresh** (link) → allows importing data faster and managing larger datasets. ▪ Check for **query parallelization** configuration settings (link) → improves query parallelization and maximizes utilization of SQL warehouse to improve overall performance. ▪ Connect Power BI to Databricks using **single sign-on (SSO)** (link) → allows leveraging security and governance controls implemented in Databricks **Unity Catalog** (link) and allows audit data access. ▪ If you need to connect to different Databricks environments, use **Power BI parameters** (link) → allows flexibility when connecting to different Databricks workspaces or different Databricks SQL warehouses. ▪ Use **gateway clusters** (link) to connect to IP ACL or private link-secured Databricks workspaces → avoid single points of failure and load balance traffic across gateways in a cluster. ▪ Use **Publish to Power BI Service** (link) → enables seamless catalog integration and data model sync, allowing you to publish datasets directly to Power BI Service without leaving the Databricks UI. ▪ Use **Automatic Publishing to Power BI** (link) → publish datasets from Unity Catalog to Power BI directly from data pipelines.	▪ **Limit the number of visuals** on each report page → limit the number of queries that will be executed. ▪ **Limit the number of rows and columns** in semantic models and report visuals → avoid large data transfers. ▪ Leverage **user-defined aggregations** (link) → improve query performance over large DirectQuery semantic models by caching pre-aggregated data. ▪ Use **automatic aggregations** (link) → continuously optimize DirectQuery semantic models by building aggregations based on Query History for maximum report performance. ▪ If referential integrity has been validated in the upstream ingestion use **"Assume Referential Integrity"** when defining table relations (link) → more efficient join strategies in SQL queries. ▪ **Avoid "many-to-many"** relationships where possible → decrease complexity and improve Power BI model efficiency. ▪ Configure **"Is nullable"** for table columns where applicable → Power BI generates simpler and more efficient SQL queries. ▪ **"Move left" transformations** whenever possible (e.g., prefer SQL views over PowerQuery transformations and DAX formulas) → leverage the power of Databricks SQL engine for more efficient report execution. ▪ If using DAX, review your code to use **efficient DAX calculations** (link) → inefficient calculations can lead to deteriorated performance. ▪ Leverage **query reduction settings** by adding Apply/Clear All Slicers button (link) → prevent a new query from being sent to the data source every time the user interacts with the report's filters. ▪ **Avoid** using **DAX calculated columns and calculated tables** in semantic models → will perform better if defined directly in your Gold tables. Measures that can be precomputed as columns also perform best if done in the Gold layer.
Troubleshooting: Why is my report slow?	1. Ensure the **data layout is regularly optimized** with VACUUM and OPTIMIZE. 2. Evaluate generating **aggregated views** for the most common and resource-intensive queries whenever applicable. 3. In the SQL Warehouse/Monitoring page (link), review the Query History looking for the queries with the longest duration: a. Ensure the query leverages the way the storage layer has been optimized or adapt it if needed. b. Open the **Query Profile**, enable verbose mode, review the tasks that took the most time and/or memory looking for: i. "*Cloud storage request duration*" to check if the cloud storage has been slow in responding. ii. "*Files read,*" "*Size of the smallest file read*" ensuring not too many small files are read. If so ensure OPTIMIZE is run regularly. iii. "*Number of output rows*" and "*Rows skipped*" ensure filters are applied sooner than later.	1. Monitor the SQL Warehouse performance in the SQL Warehouse/Monitoring page (link) looking for: a. **Running queries vs. Queued queries.** If too many Queued queries: 1) increase the number of clusters if already reaching the maximum allowed, 2) evaluate if worth increasing the cluster size if there are long-running queries. b. Number of **active clusters**: 1) evaluate to set a higher minimum number of clusters if queries are queued while scaling out, 2) check if the cluster scaled in to zero when the queries arrived. 3. Monitor **Query History** in the SQL Warehouse/Monitoring page (link), reviewing query details for: a. "*Scheduling*" (i.e., time spent in queue) vs. "*Optimizing Query*" (i.e., time spent mainly identifying data to be skipped) vs. "*Executing*" (i.e., time spent executing the query). b. "*Result fetching by client*" (i.e., time spent for the client to download the result set). c. "*Rows returned*" to ensure the query does not return too many rows. d. "*Bytes read from cache*" to evaluate the disk cache efficiency. e. "*Bytes spilled to disk*" to ensure no data is spilled to disk; if so, increase the cluster size.	1. Evaluate which Power BI storage mode is used. Prefer **DirectQuery for Fact tables** and **Dual for Dimensions** → ideal compromise between performance and freshness of the data. 2. For very **small, static and performance-sensitive** (<2s) reports, evaluate the usage of **Import** mode → would provide the best report performance. 3. Especially for DirectQuery, check how many queries Power BI can send **in parallel** to Databricks. Ensure the Databricks **SQL warehouse is sized accordingly** to handle the required level of parallelism → avoids queries to be queued, resulting in a slow report. 4. For performance fine-tuning, evaluate the following properties of Power BI semantic models: a. *"Maximum connections per data source"* b. *"Maximum number of simultaneous evaluations"* c. *"Maximum number of concurrent jobs"* d. *"MaxParallelismPerQuery"* 5. Monitor the queries in the SQL Warehouse/Monitoring page, looking at "*Started at*" and at which **time the queries arrived** to validate the effective parallelism in Power BI.	1. Use **Power BI Performance Analyzer** to examine report element performance (link) → identify the visual that takes the most to load and where the bottleneck is (DAX Query, Visual Display, DirectQuery, etc.). 2. Ensure there are **not too many visuals** in the same report → many visuals could generate lots of queries that can be queued by Power BI or Databricks (e.g., the query itself runs fast but spends time in the queue). 3. For the most common and resource-intensive queries, **ask for creating SQL views or persisted tables** in the Gold layer, which provides **pre-aggregated data** (often valid for Date dimension) → results in overall better performance. 4. Ensure there are no SQL queries returning large result sets (1,000s of records), which is often an indication of inefficient DAX formulas (e.g., TOPN function) → evaluate the complexity of DAX formulas and optimize where possible.
What you should read	▪ Power Up With Power BI and Lakehouse in Azure Databricks: Part 3 — Tuning Azure Databricks SQL (link) ▪ Star Schema Data Modeling Best Practices on Databricks SQL (link) ▪ One Big Table vs. Dimensional Modeling on Databricks SQL (link)	▪ SQL Warehouse Sizing, Scaling and Queuing Behavior (link) ▪ DBSQL Warehouse Advisor (link) ▪ Tune Query Performance in Databricks SQL With the Query Profile (link)	▪ Power Up Your BI With Microsoft Power BI and Lakehouse in Azure Databricks: Part 1 — Essentials (link) ▪ Power BI — Databricks SQL QuickStart Samples (link) ▪ Power BI on DBSQL Design Patterns (link) ▪ Boosting Power BI Performance With Azure Databricks Through Automatic Aggregations (link)	▪ Power Up Your BI With Microsoft Power BI and Lakehouse in Azure Databricks: Part 2 — Tuning Power BI (link) ▪ Optimization Guide for Power BI (link) ▪ Monitor Report Performance in Power BI (link) ▪ Troubleshoot Report Performance in Power BI (link)