Cloud Data Warehouse Cost Optimization

Reduce Your Cloud Data Warehouse Costs by 30-60% with Proven Optimization Strategies

Version: 1.0 | **Updated:** Jan 2026

www.EnterprisedDataSolutions.co.nz
Contact@EnterprisedDataSolutions.co.nz

Table of Contents

Executive Summary

Expected Cost Reduction Timeline

Month 1	Month 2-3	Month 4-6	Total
Quick Wins	Query/Storage	Advanced	6-Month
25-35%	**+15-25%**	**+10-15%**	**50-75%**

Cloud data warehouses offer incredible scalability and performance, but costs can quickly spiral out of control without proper optimization. Organizations typically overspend by 40-70% on cloud data warehouse infrastructure due to:

- Oversized compute resources running 24/7
- Inefficient query patterns scanning unnecessary data
- Poor data organization and clustering
- Lack of lifecycle policies for cold data
- Missing cost visibility and accountability

This guide provides actionable strategies to reduce cloud data warehouse costs by 30-60% while maintaining or improving performance.

Key Strategies

Strategy	Typical Savings	Implementation Effort	Time to Value
Right-size warehouses	20-40%	Low	Immediate
Implement auto-suspend/resume	30-50%	Low	Immediate
Optimize query patterns	15-35%	Medium	1-2 weeks
Implement clustering/partitioning	20-40%	Medium	2-4 weeks
Storage tier optimization	40-60%	Low	Immediate
Data lifecycle policies	20-40%	Medium	2-4 weeks
Query result caching	10-30%	Low	Immediate
Materialized views	15-35%	Medium	1-2 weeks

Introduction

Cloud Data Warehouse Pricing Models

SNOWFLAKE	BIGQUERY	REDSHIFT	SYNAPSE
Compute + Storage	On-demand Flat-rate	Instance-based	DWU Serverless

The Cloud Data Warehouse Cost Challenge

Cloud data warehouses like Snowflake, BigQuery, Redshift, and Azure Synapse have revolutionized data analytics with on-demand scalability and near-infinite storage. However, this flexibility comes with a cost model that can quickly become expensive if not properly managed.

Common Cost Scenarios

Scenario 1: The Always-On Warehouse

Actual usage: 8 hours/day (business hours)
Cost: $17,520/month
Optimized cost: $5,840/month with auto-suspend
Savings: $11,680/month (67%)

67% saved

Scenario 2: The Full-Scan Query

Cost: $2,500/month (BigQuery on-demand)
Optimized cost: $50/month with partitioning
Savings: $2,450/month (98%)

98% saved

Scenario 3: The Retention Hoarder

Cost: $11,500/month
Optimized cost: $2,800/month with tiering
Savings: $8,700/month (76%)

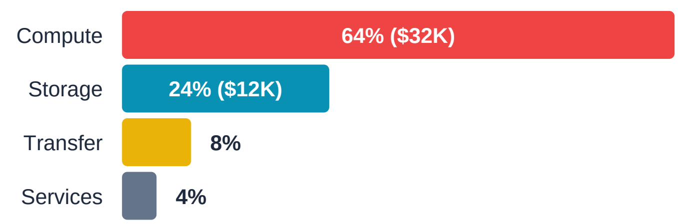
76% saved

Cloud Data Warehouse Cost Models

Platform	Pricing Model	Key Cost Drivers
Snowflake	Compute + Storage	Warehouse runtime, storage volume, data transfer
BigQuery	On-demand or Flat-rate	Bytes processed (on-demand), slot hours (flat-rate), storage
Redshift	Instance-based	Node hours, storage, Spectrum queries
Synapse	DWU hours or Serverless	DWU hours, storage, queries processed

Understanding Cloud Data Warehouse Costs

$50K/Month Typical Cost Breakdown



Optimization Opportunities

From the above breakdown:

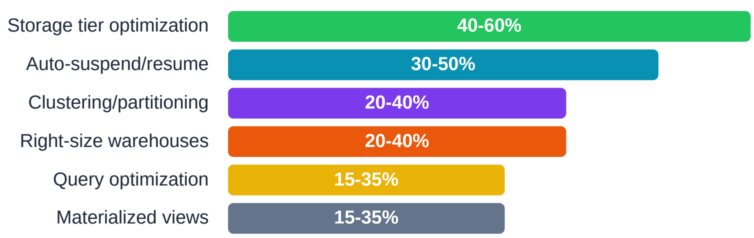
- **Compute (64%)**: Reduce with auto-suspend, right-sizing, query optimization
- **Storage (24%)**: Reduce with lifecycle policies, compression, deduplication
- **Data Transfer (8%)**: Reduce with result caching, materialized views
- **Cloud Services (4%)**: Usually < 10% of compute, minimal optimization needed

Potential savings:

- Compute: 40% reduction = $12,800/month
- Storage: 50% reduction = $6,000/month
- Data Transfer: 30% reduction = $1,200/month
- **Total: $20,000/month (40% overall reduction)**

Cost Components Breakdown

Optimization Strategies: Typical Savings



1. Compute Costs

What drives compute costs:

- Warehouse size (Small, Medium, Large, X-Large, etc.)
- Runtime hours
- Number of concurrent warehouses
- Scaling frequency

Snowflake compute pricing:

Warehouse Size	Credits/Hour	$/Hour (Standard)	$/Hour (Enterprise)
Small	1	$2	$3
Medium	2	$4	$6
Large	4	$8	$12
X-Large	8	$16	$24
2X-Large	16	$32	$48
3X-Large	32	$64	$96
4X-Large	64	$128	$192

Calculation:

```
Monthly cost = Warehouse size credits × $/credit × hours run
Example: X-Large running 24/7
= 8 credits/hour × $3/credit × 730 hours/month
= $17,520/month
```

2. Storage Costs

What drives storage costs:

- Volume of data stored (compressed)
- Data retention period
- Fail-safe and time travel settings
- Number of table copies/clones

Pricing comparison:

Platform	Storage Cost/TB/Month	Compression Ratio
Snowflake	$23 (+ $23 fail-safe)	3-5x
BigQuery	$20 (active)	2-4x
$10 (long-term)		
Redshift	$24 (RA3)	3-4x
Synapse	$23	3-5x

3. Data Transfer Costs

What drives data transfer costs:

- Egress to external systems
- Cross-region data movement
- Replication and backup

Typical costs:

```
Data Transfer Type | Cost
Ingress (upload) | Free
Egress within region | Free (or minimal)
Egress cross-region | $0.02 - $0.12/GB
Egress to internet | $0.05 - $0.15/GB
```

Snowflake Cost Optimization

Strategy 1: Auto-Suspend and Auto-Resume

Problem: Warehouses left running when not in use.

Solution:

```
-- Set auto-suspend to 5 minutes (300 seconds)
ALTER WAREHOUSE ETL_WH SET AUTO_SUSPEND = 300;

-- Enable auto-resume
ALTER WAREHOUSE ETL_WH SET AUTO_RESUME = TRUE;

-- For dev warehouses, even more aggressive
ALTER WAREHOUSE DEV_WH SET AUTO_SUSPEND = 60;
```

Calculation:

```
Scenario: Large warehouse used 8 hours/day, 5 days/week

Before (always-on):
730 hours/month × 4 credits/hour × $3 = $8,760/month

After (auto-suspend):
8 hours/day × 22 days/month = 176 hours
176 hours × 4 credits/hour × $3 = $2,112/month

Savings: $6,648/month (76%)
```

Strategy 2: Right-Size Warehouses

Problem: Using oversized warehouses for workloads.

Solution:

```
-- Monitor warehouse load
SELECT
warehouse_name,
AVG(avg_running) as avg_queries_running,
AVG(avg_queued_load) as avg_queued,
COUNT(*) as measurements
FROM snowflake.account_usage.warehouse_load_history
WHERE start_time >= DATEADD(day, -7, CURRENT_TIMESTAMP())
GROUP BY warehouse_name;

-- Right-sizing rules:
-- avg_queries_running < 1.0: Downsize by 1 level
-- avg_queued > 0: Upsize by 1 level
-- avg_queued > 5: Consider multi-cluster

-- Example: Downsize from Large to Medium
ALTER WAREHOUSE REPORTING_WH SET WAREHOUSE_SIZE = MEDIUM;
```

Impact:

Strategy 3: Multi-Cluster Warehouses

Problem: Over-provisioning single large warehouse for peak load.

Solution:

Strategy 4: Query Result Caching

Problem: Re-running identical queries wastes compute.

Solution:

```
-- Enable result caching (default, but verify)
ALTER WAREHOUSE ANALYTICS_WH SET STATEMENT_TIMEOUT_IN_SECONDS = 7200;

-- Results cached for 24 hours if:
-- 1. Exact same SQL
-- 2. Underlying data unchanged
-- 3. Within 24-hour window

-- Monitor cache hit rate:
SELECT
warehouse_name,
SUM(CASE WHEN query_id IS NOT NULL THEN 1 ELSE 0 END) as total_queries,
SUM(CASE WHEN partitions_scanned = 0 THEN 1 ELSE 0 END) as cached_queries,
(cached_queries / total_queries * 100) as cache_hit_rate
FROM snowflake.account_usage.query_history
WHERE start_time >= DATEADD(day, -7, CURRENT_TIMESTAMP())
GROUP BY warehouse_name;

-- Target: > 20% cache hit rate
```

Impact:

```
1000 queries/day, 25% cache hit rate

Before caching:
1000 queries × 5 seconds avg × 8 credits/hour = 11.1 credit hours/day

After caching:
750 queries × 5 seconds avg × 8 credits/hour = 8.3 credit hours/day

Savings: 2.8 credit hours/day × $3 × 30 days = $252/month
```

Strategy 5: Clustering and Partitioning

Problem: Queries scan entire large tables.

Solution:

```
-- Identify high-cost queries
SELECT
query_text,
warehouse_name,
total_elapsed_time,
bytes_scanned,
partitions_scanned,
partitions_total,
(partitions_scanned::float / partitions_total * 100) as scan_efficiency
FROM snowflake.account_usage.query_history
WHERE start_time >= DATEADD(day, -7, CURRENT_TIMESTAMP())
AND total_elapsed_time > 60000 -- > 1 minute
ORDER BY bytes_scanned DESC
LIMIT 100;

-- Add clustering key to large tables
ALTER TABLE events CLUSTER BY (event_date, event_type);

-- Verify clustering effectiveness
SELECT SYSTEM$CLUSTERING_INFORMATION('events', '(event_date, event_type)');

-- Auto-clustering (recommended for Snowflake Enterprise)
```

```
ALTER TABLE events RESUME RECLUSTER;
```

Impact:

```
Query scanning 500GB table daily

Before clustering:
Partitions scanned: 100% (500GB)
Query cost: 500GB scan

After clustering on date column:
Partitions scanned: 2% (10GB) for typical date-filtered queries
Query cost: 10GB scan

Savings: 98% reduction in data scanned
If 100 queries/day: ~$3,000/month saved
```

Strategy 6: Storage Optimization

Problem: Paying for unnecessary data retention.

Solution:

Impact:

Strategy 7: Resource Monitors

Problem: No cost guardrails, surprise bills.

Solution:

```
-- Create account-level monitor
CREATE RESOURCE MONITOR account_monthly_limit WITH
CREDIT_QUOTA = 10000
FREQUENCY = MONTHLY
START_TIMESTAMP = IMMEDIATELY
TRIGGERS
ON 75 PERCENT DO NOTIFY
ON 90 PERCENT DO SUSPEND
ON 100 PERCENT DO SUSPEND_IMMEDIATE;

-- Apply to account
ALTER ACCOUNT SET RESOURCE_MONITOR = account_monthly_limit;

-- Create warehouse-specific monitors
CREATE RESOURCE MONITOR etl_warehouse_limit WITH
CREDIT_QUOTA = 2000
FREQUENCY = MONTHLY
START_TIMESTAMP = IMMEDIATELY
TRIGGERS ON 90 PERCENT DO SUSPEND;

ALTER WAREHOUSE ETL_WH SET RESOURCE_MONITOR = etl_warehouse_limit;
```

BigQuery Cost Optimization

Pricing Model

BigQuery offers two pricing models:

On-Demand:

- $5 per TB processed (first 1TB/month free)
- Pay only for queries run
- No upfront costs

Flat-Rate (Capacity):

- Buy slots (units of computational capacity)
- 100 slots = $2,000/month
- Unlimited queries within slot capacity
- Better for high query volume

Break-even analysis:

```
On-demand cost = TB processed × $5
Flat-rate cost = Slots × $20/month

Example:
- Processing 500TB/month on-demand: $2,500
- 100 slots flat-rate: $2,000
- Savings: $500/month with flat-rate (+ unlimited queries)

Rule of thumb: Flat-rate if processing > 400TB/month
```

Strategy 1: Partitioning

Problem: Queries scan entire tables.

Solution:

```
-- Create partitioned table (by date)
CREATE TABLE events_partitioned
PARTITION BY DATE(event_timestamp)
AS SELECT * FROM events;

-- Partition by integer range (for large dimension tables)
CREATE TABLE users_partitioned
PARTITION BY RANGE_BUCKET(user_id, GENERATE_ARRAY(0, 100000000, 100000))
AS SELECT * FROM users;

-- Query with partition filter
SELECT COUNT(*) FROM events_partitioned
WHERE DATE(event_timestamp) = '2025-01-15';
-- Scans only 1 day's partition, not entire table
```

Impact:

```
Table: 10TB, 3 years of data
Query: Analyze last 30 days

Before partitioning:
Bytes processed: 10TB
Cost: 10TB × $5 = $50 per query

After partitioning:
Bytes processed: ~83GB (30 days of 10TB / 1095 days)
Cost: 0.083TB × $5 = $0.42 per query
```

```
Per query savings: $49.58 (99.2% reduction)
100 queries/day: $4,958/day = $148,740/month saved!
```

Strategy 2: Clustering

Problem: Even partitioned tables scan too much data.

Solution:

```
-- Add clustering to partitioned table
CREATE TABLE events_clustered
PARTITION BY DATE(event_timestamp)
CLUSTER BY user_id, event_type
AS SELECT * FROM events;

-- Query benefits from clustering
SELECT * FROM events_clustered
WHERE DATE(event_timestamp) BETWEEN '2025-01-01' AND '2025-01-31'
AND user_id = 12345
AND event_type = 'purchase';
-- Partition filter + cluster filter = minimal data scanned
```

Impact:

Strategy 3: Materialized Views

Problem: Complex aggregations run repeatedly.

Solution:

```
-- Create materialized view for common aggregation
CREATE MATERIALIZED VIEW daily_revenue_by_product AS
SELECT
DATE(order_timestamp) as order_date,
product_id,
SUM(revenue) as total_revenue,
COUNT(*) as order_count
FROM orders
GROUP BY order_date, product_id;

-- Query materialized view instead of base table
SELECT * FROM daily_revenue_by_product
WHERE order_date >= '2025-01-01';
-- Scans small pre-aggregated table, not large orders table
```

Impact:

```
Query: Daily revenue aggregation

Base table query:
- Scans: 5TB orders table
- Cost: $25 per query
- Latency: 45 seconds

Materialized view query:
- Scans: 50GB materialized view
- Cost: $0.25 per query
- Latency: 2 seconds

Savings: $24.75 per query (99%)
100 queries/day: $2,475/day saved!
```

Strategy 4: Query Result Caching

Problem: Identical queries re-process data.

Solution:

```
-- BigQuery automatically caches results for 24 hours
-- Ensure queries are identical (whitespace matters!)

-- Use parameterized queries for caching
DECLARE target_date DATE DEFAULT '2025-01-15';

SELECT COUNT(*) FROM events
WHERE DATE(event_timestamp) = target_date;

-- Cached if:
-- 1. Exact same SQL (byte-for-byte)
-- 2. Tables haven't changed
-- 3. Within 24 hours
-- 4. Not using non-deterministic functions (CURRENT_TIMESTAMP, RAND, etc.)

-- Monitor cache hits (check bytes billed vs bytes processed)
```

Impact:

Strategy 5: Optimize Query Patterns

Problem: Inefficient SQL scans unnecessary data.

Solutions:

5a. SELECT only needed columns

```
-- Bad: SELECT *
SELECT * FROM large_table WHERE date = '2025-01-15';
-- Scans all 50 columns

-- Good: SELECT specific columns
SELECT user_id, event_type, timestamp FROM large_table WHERE date = '2025-01-15';
-- Scans only 3 columns, 94% less data
```

5b. Use LIMIT for exploration

```
-- Bad: Explore without LIMIT
SELECT * FROM huge_table WHERE category = 'electronics';
-- Scans entire table

-- Good: Use LIMIT for exploration
SELECT * FROM huge_table WHERE category = 'electronics' LIMIT 100;
-- Still scans entire table in BigQuery!

-- Better: Partition + LIMIT
SELECT * FROM huge_table
WHERE DATE(created_at) = CURRENT_DATE()
AND category = 'electronics'
LIMIT 100;
-- Scans only today's partition
```

5c. Avoid SELECT DISTINCT on large datasets

```
-- Bad: DISTINCT scans all data
SELECT DISTINCT user_id FROM events;
-- Scans entire 10TB table

-- Good: Use GROUP BY with partitioning
SELECT user_id FROM events
WHERE DATE(event_timestamp) >= DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY)
GROUP BY user_id;
-- Scans only 30 days
```

5d. Push down filters

```
-- Bad: Filter after JOIN
SELECT o.* FROM orders o
```

```
JOIN users u ON o.user_id = u.user_id
WHERE o.order_date = '2025-01-15';
-- Joins all users, then filters

-- Good: Filter before JOIN
SELECT o.* FROM
(SELECT * FROM orders WHERE order_date = '2025-01-15') o
JOIN users u ON o.user_id = u.user_id;
-- Filters first, joins small subset
```

Strategy 6: Storage Optimization

Problem: Paying $20/TB for rarely accessed data.

Solution:

Impact:

```
Scenario: 20TB of 3+ year old data

Active storage cost:
20TB × $20/TB = $400/month

Options:
1. Long-term storage (automatic after 90 days):
20TB × $10/TB = $200/month
Savings: $200/month (50%)

2. Archive to GCS + external table:
GCS storage: 20TB × $2/TB = $40/month
Query cost: Only when accessed (~$5/query for full scan)
Savings: $360/month (90%) if rarely queried
```

Strategy 7: Flat-Rate Pricing for High Volume

Problem: High on-demand costs for predictable workloads.

Solution:

```
-- Calculate break-even point
-- On-demand: $5/TB
-- Flat-rate: 100 slots = $2,000/month

-- If processing > 400TB/month, flat-rate is cheaper
-- Example: 600TB/month

On-demand: 600TB × $5 = $3,000/month
Flat-rate: $2,000/month (100 slots)
Savings: $1,000/month (33%)

-- Plus: Unlimited queries, predictable costs, priority access

-- Implement flex slots for variable workloads
-- Commit: 100 slots ($2,000/month)
-- Flex: +50 slots as needed ($1,000/month when used)
-- Total: $2,000-3,000/month vs $3,000+ on-demand
```

Redshift Cost Optimization

Pricing Model

Redshift offers multiple node types:

RA3 (Recommended):

- Managed storage (separate from compute)
- $3.26/hour for ra3.4xlarge (12 vCPU, 96GB RAM)
- Storage: $0.024/GB/month
- Scales compute and storage independently

DC2:

- SSD-based (storage coupled with compute)
- $1.086/hour for dc2.large (2 vCPU, 15GB RAM)
- Limited to node storage capacity

Strategy 1: Right-Size Cluster

Problem: Over-provisioned cluster running 24/7.

Solution:

Impact:

Strategy 2: Pause and Resume

Problem: Dev/test clusters running when not needed.

Solution:

```
-- Pause cluster (via AWS CLI)
aws redshift pause-cluster --cluster-identifier dev-cluster

-- Resume cluster
aws redshift resume-cluster --cluster-identifier dev-cluster

-- Automate with Lambda:
-- Pause at 6 PM, resume at 8 AM on weekdays
```

Impact:

```
Dev cluster: 2 × ra3.xlplus = 2 × $1.63/hour

Before (always-on):
730 hours/month × $3.26 = $2,380/month

After (10 hours/day, 5 days/week):
~220 hours/month × $3.26 = $717/month

Savings: $1,663/month (70%)
```

Strategy 3: Distribution and Sort Keys

Problem: Queries involve massive data shuffling.

Solution:

Impact:

```
Query: JOIN orders (1B rows) with customers (10M rows) on customer_id

Before (no distribution key):
- Shuffle: 500GB across network
- Query time: 180 seconds
- Cost: High I/O, network saturation

After (DISTKEY customer_id on both tables):
- Shuffle: Minimal (co-located data)
- Query time: 12 seconds
- Cost: 93% faster, 93% less I/O

For 1000 queries/day: Saves ~47 hours of compute daily
```

Strategy 4: Workload Management (WLM)

Problem: Resource contention between workloads.

Solution:

```
-- Configure WLM queues for different workloads
-- Parameter group settings:

Queue 1 (ETL): 40% memory, concurrency 3
Queue 2 (Reporting): 30% memory, concurrency 5
Queue 3 (Ad-hoc): 20% memory, concurrency 8
Queue 4 (Superuser): 10% memory, concurrency 1

-- Short query acceleration (SQA)
-- Automatically prioritizes queries < 20 seconds

-- Result: No resource starvation, predictable performance
```

Strategy 5: Spectrum for Cold Data

Problem: Storing rarely accessed data in Redshift.

Solution:

```
-- Archive old data to S3
UNLOAD ('SELECT * FROM events WHERE event_year < 2022')
TO 's3://my-bucket/archive/events/'
FORMAT AS PARQUET
PARALLEL ON;

-- Create Spectrum external table
CREATE EXTERNAL TABLE spectrum.events_archive (
event_id BIGINT,
event_date DATE,
user_id BIGINT,
event_type VARCHAR(50)
)
STORED AS PARQUET
LOCATION 's3://my-bucket/archive/events/';

-- Query when needed (pay only for S3 data scanned)
SELECT COUNT(*) FROM spectrum.events_archive
WHERE event_year = 2021;

-- Drop original table
DROP TABLE events_archive;
```

Impact:

```
Scenario: 10TB of cold data (2+ years old)

Redshift RA3 storage cost:
10,000GB × $0.024/GB = $240/month

S3 Standard-IA cost:
10,000GB × $0.0125/GB = $125/month
+ Spectrum scan cost: ~$5/TB scanned = $50/month (if query 10TB monthly)

Total S3 + Spectrum: $175/month
Savings: $65/month (27%)

If querying less frequently: Up to 75% savings
```

Strategy 6: Automatic Workload Management (Auto WLM)

Problem: Manual WLM configuration is complex.

Solution:

```
-- Enable Auto WLM (Redshift console or parameter group)
-- Benefits:
-- - Dynamic memory allocation
-- - Automatic query prioritization
-- - Machine learning-based optimization
-- - Reduces manual tuning

-- Monitor Auto WLM performance
SELECT
service_class,
num_queued_queries,
avg_queue_time,
avg_execution_time
FROM svl_query_metrics
WHERE service_class > 4 -- User queues
AND date_trunc('day', start_time) = CURRENT_DATE
GROUP BY service_class;
```

Strategy 7: Concurrency Scaling

Problem: Query queuing during peak times.

Solution:

Impact:

```
Scenario: Peak load 2 hours/day needs 2× capacity

Option 1: 2× main cluster size always
Cost: 2× base cost = $38,156/month (8 nodes vs 4)

Option 2: Concurrency scaling
- Base cluster: 4 nodes = $19,078/month
- Scaling: 2 hours/day - 1 free hour = 1 hour/day × 30 days × $13.04/hour = $391/month
- Total: $19,469/month

Savings: $18,687/month (49%)
```

Azure Synapse Cost Optimization

Pricing Model

Dedicated SQL Pool (formerly SQL DW):

- DWU-based: $1.20/hour per 100 DWU (DW100c)
- Scales: DW100c to DW30000c
- Pay for provisioned capacity

Serverless SQL Pool:

- $5 per TB processed
- No provisioned capacity
- Pay per query (similar to BigQuery on-demand)

Strategy 1: Pause and Resume Dedicated Pools

Problem: Dedicated pools running 24/7.

Solution:

```
-- Pause pool (PowerShell)
Suspend-AzSqlDatabase -ResourceGroupName "myResourceGroup" `
-ServerName "myServer" -DatabaseName "myDataWarehouse"

-- Resume pool
Resume-AzSqlDatabase -ResourceGroupName "myResourceGroup" `
-ServerName "myServer" -DatabaseName "myDataWarehouse"

-- Automate with Azure Automation:
-- Schedule: Pause at 7 PM, resume at 7 AM weekdays
```

Impact:

```
DW500c (500 DWU) = $6/hour

Before (always-on):
730 hours/month × $6 = $4,380/month

After (12 hours/day, 5 days/week):
~260 hours/month × $6 = $1,560/month

Savings: $2,820/month (64%)
```

Strategy 2: Scale DWUs Dynamically

Problem: Over-provisioned for average workload.

Solution:

```
-- Scale up for ETL (PowerShell)
Set-AzSqlDatabase -ResourceGroupName "myResourceGroup" `
-ServerName "myServer" -DatabaseName "myDataWarehouse" `
-RequestedServiceObjectiveName "DW1000c"

-- Scale down after ETL
Set-AzSqlDatabase -ResourceGroupName "myResourceGroup" `
-ServerName "myServer" -DatabaseName "myDataWarehouse" `
-RequestedServiceObjectiveName "DW500c"

-- Automate scaling:
-- 7 AM-9 AM (ETL): DW1000c
-- 9 AM-5 PM (reporting): DW500c
```

```
-- 5 PM-7 PM (batch jobs): DW1000c
```

Impact:

```
Workload:
- ETL (2 hours/day): Needs DW1000c ($12/hour)
- Reporting (8 hours/day): Needs DW500c ($6/hour)
- Batch (2 hours/day): Needs DW1000c ($12/hour)

Before (always DW1000c for 12 hours/day):
12 hours × $12 × 22 days = $3,168/month

After (dynamic scaling):
- ETL: 2 hours × $12 × 22 days = $528
- Reporting: 8 hours × $6 × 22 days = $1,056
- Batch: 2 hours × $12 × 22 days = $528
- Total: $2,112/month

Savings: $1,056/month (33%)
```

Strategy 3: Use Serverless for Ad-Hoc Queries

Problem: Paying for dedicated pool for occasional queries.

Solution:

```
-- Use serverless SQL pool for:
-- - Exploration and development
-- - Infrequent analytical queries
-- - Data lake queries

-- Create external table on serverless pool
CREATE EXTERNAL TABLE events_external
WITH (
LOCATION = 'events/',
DATA_SOURCE = AzureDataLakeStorage,
FILE_FORMAT = ParquetFormat
)
AS
SELECT * FROM OPENROWSET(
BULK 'events/*.parquet',
DATA_SOURCE = 'AzureDataLakeStorage',
FORMAT = 'PARQUET'
) AS events;

-- Query pays only for data processed
SELECT COUNT(*) FROM events_external
WHERE event_date = '2025-01-15';
```

Impact:

```
Ad-hoc queries:
- 50 queries/month
- 100GB avg per query
- Total: 5TB/month processed

Dedicated pool (DW100c, 2 hours/month):
2 hours × $1.20 = $2.40/month
(But pool needs to be available, so likely higher)

Serverless:
5TB × $5/TB = $25/month

If queries are truly ad-hoc (<10 queries/month): Serverless cheaper
If regular workload: Dedicated pool cheaper

Recommendation: Use serverless for <100TB/month ad-hoc queries
```

Strategy 4: Result Set Caching

Problem: Repeated queries reprocess data.

Solution:

```
-- Enable result caching (database level)
ALTER DATABASE myDataWarehouse
SET RESULT_SET_CACHING ON;

-- Enable for session
SET RESULT_SET_CACHING ON;

-- Check cache hit rate
SELECT
query_hash,
COUNT(*) as execution_count,
SUM(CASE WHEN result_cache_hit = 1 THEN 1 ELSE 0 END) as cache_hits,
SUM(CASE WHEN result_cache_hit = 1 THEN 1 ELSE 0 END)::float / COUNT(*) as cache_hit_rate
FROM sys.dm_pdw_exec_requests
WHERE status = 'Completed'
AND submit_time >= DATEADD(day, -7, GETDATE())
GROUP BY query_hash
HAVING COUNT(*) > 10
ORDER BY execution_count DESC;

-- Target: >25% cache hit rate for dashboards
```

Impact:

```
Dashboard: 500 queries/day, 30% cache hit rate

Before caching:
500 queries × 10 seconds avg DW500c time = 5,000 seconds/day
= 1.39 hours/day = 30.5 hours/month
Cost: 30.5 hours × $6 = $183/month

After caching:
350 queries × 10 seconds avg = 3,500 seconds/day
= 0.97 hours/day = 21.3 hours/month
Cost: 21.3 hours × $6 = $128/month

Savings: $55/month (30%)
```

Strategy 5: Workload Management

Problem: Resource contention between workloads.

Solution:

Universal Optimization Strategies

These strategies apply across all cloud data warehouse platforms:

Strategy 1: Incremental Processing

Problem: Full table refreshes waste compute.

Solution:

```
-- Bad: Full refresh daily
TRUNCATE TABLE daily_summary;
INSERT INTO daily_summary
SELECT * FROM process_all_data();
-- Scans entire history

-- Good: Incremental processing
DELETE FROM daily_summary WHERE process_date >= CURRENT_DATE - 1;
INSERT INTO daily_summary
SELECT * FROM process_incremental_data(CURRENT_DATE - 1);
-- Processes only yesterday's data

-- Use change data capture (CDC) when possible
-- Snowflake: CHANGES clause
-- BigQuery: _PARTITIONTIME pseudo-column
-- Redshift: Timestamp-based incremental
```

Impact:

```
Full refresh:
- Processes: 10TB daily
- Time: 2 hours
- Cost: $100/day

Incremental:
- Processes: 50GB daily (new data)
- Time: 10 minutes
- Cost: $2.50/day

Savings: $97.50/day = $2,925/month (97.5%)
```

Strategy 2: Compression

Problem: Storing uncompressed or poorly compressed data.

Solution:

```
-- Snowflake: Automatic compression (nothing to configure)

-- BigQuery: Use efficient data types
CREATE TABLE events_optimized (
event_id INT64, -- Not STRING
event_timestamp TIMESTAMP, -- Not STRING
amount NUMERIC(10, 2), -- Not FLOAT64 for money
country STRING -- OK for low-cardinality text
);

-- Redshift: Use compression encodings
CREATE TABLE events (
event_id BIGINT ENCODE az64, -- Monotonic IDs
event_type VARCHAR(50) ENCODE lzo, -- Text
amount DECIMAL(10,2) ENCODE az64, -- Numeric
event_timestamp TIMESTAMP ENCODE az64 -- Timestamps
);

-- Or let Redshift auto-analyze:
ANALYZE COMPRESSION events;
```

```
-- Then recreate table with suggested encodings
```

Impact:

Strategy 3: Deduplication

Problem: Duplicate data from poor ETL processes.

Solution:

Impact:

Warehouse Sizing and Auto-Scaling

Right-Sizing Framework

Step 1: Measure current utilization

```
-- Snowflake
SELECT
warehouse_name,
AVG(avg_running) as avg_concurrent_queries,
MAX(avg_running) as peak_concurrent_queries,
AVG(avg_queued_load) as avg_queue_depth
FROM snowflake.account_usage.warehouse_load_history
WHERE start_time >= DATEADD(day, -30, CURRENT_TIMESTAMP())
GROUP BY warehouse_name;

-- Rules:
-- avg_concurrent < 0.5: Downsize 1-2 levels
-- avg_concurrent < 1.0: Downsize 1 level
-- avg_queue_depth > 0: Consider upsize or multi-cluster
-- peak_concurrent >> avg_concurrent: Use multi-cluster
```

Step 2: Calculate optimal size

```
Optimal size = CEIL(Peak concurrent queries / Queries per size tier)

Example:
- Peak concurrent: 12 queries
- Queries per tier (rule of thumb):
- Small: 1-2 queries
- Medium: 2-4 queries
- Large: 4-8 queries
- X-Large: 8-16 queries

Optimal: Large warehouse (handles 4-8 concurrent)
Or: Medium multi-cluster (2-4 per cluster × 4 clusters max = 16)

Recommendation: Medium multi-cluster (lower cost, better elasticity)
```

Step 3: Implement and monitor

```
-- Start conservative
ALTER WAREHOUSE ANALYTICS_WH SET WAREHOUSE_SIZE = MEDIUM;

-- Monitor for 1 week
-- If queue depth > 0: Upsize or add multi-cluster
-- If utilization < 50%: Downsize
```

Auto-Scaling Best Practices

Snowflake multi-cluster:

```
CREATE WAREHOUSE ANALYTICS_WH WITH
WAREHOUSE_SIZE = 'MEDIUM'
MIN_CLUSTER_COUNT = 1
MAX_CLUSTER_COUNT = 4
SCALING_POLICY = 'STANDARD' -- Conservative
-- Or 'ECONOMY' for more aggressive scaling
AUTO_SUSPEND = 300
AUTO_RESUME = TRUE;

-- STANDARD: Starts additional cluster immediately when queue forms
-- ECONOMY: Waits ~6 minutes before starting additional cluster

-- Use STANDARD for user-facing queries
-- Use ECONOMY for batch jobs where latency is acceptable
```

Redshift concurrency scaling:

```
-- Enable in workload management (WLM) console
-- Max concurrency scaling clusters: 3

-- Monitor and tune
SELECT
service_class,
num_concurrency_scaling_requests,
avg_concurrency_scaling_seconds
FROM svl_concurrency_scaling_usage
WHERE start_time >= CURRENT_DATE - 30;

-- Adjust max clusters based on usage patterns
```

Query Optimization

Query Performance Hierarchy

```
Impact on Cost (Highest to Lowest):

1. Data scanned volume (100x impact)
- Full table scan vs partition scan
- SELECT * vs SELECT specific columns

2. Query complexity (10x impact)
- Nested subqueries vs CTEs
- Cartesian joins vs indexed joins

3. Warehouse size (2-4x impact)
- X-Large vs Medium

4. Result caching (100% when cached)
- Cache hit vs cache miss
```

Optimization Checklist

Before optimization:

```
-- Bad query (scans 5TB, takes 2 minutes)
SELECT *
FROM events e
JOIN users u ON e.user_id = u.user_id
WHERE e.event_type = 'purchase'
AND YEAR(e.event_timestamp) = 2025;
```

Apply optimizations:

1. Partition filter

```
-- Add explicit date range instead of YEAR function
WHERE e.event_timestamp >= '2025-01-01'
AND e.event_timestamp < '2026-01-01'
-- Enables partition pruning
```

2. Select only needed columns

```
-- Instead of SELECT *
SELECT
e.event_id,
e.event_timestamp,
u.email,
e.amount
```

3. Pre-filter before join

```
-- Filter events before joining
FROM (
SELECT event_id, event_timestamp, user_id, amount
FROM events
WHERE event_timestamp >= '2025-01-01'
AND event_timestamp < '2026-01-01'
AND event_type = 'purchase'
) e
JOIN users u ON e.user_id = u.user_id
```

Final optimized query:

```
SELECT
e.event_id,
e.event_timestamp,
u.email,
```

```
e.amount
FROM (
SELECT event_id, event_timestamp, user_id, amount
FROM events
WHERE event_timestamp >= '2025-01-01'
AND event_timestamp < '2026-01-01'
AND event_type = 'purchase'
) e
JOIN users u ON e.user_id = u.user_id;

-- Scans: 100GB (vs 5TB)
-- Time: 3 seconds (vs 2 minutes)
-- Cost: 98% reduction
```

Storage Tier Optimization

Storage Lifecycle Strategy

Tier 1: Hot (Active Storage)

- Data: Last 3-6 months
- Access: Daily
- Cost: Highest ($20-24/TB/month)
- Performance: Fastest

Tier 2: Warm (Long-term Storage)

- Data: 6 months - 2 years
- Access: Weekly/monthly
- Cost: Medium ($10-12/TB/month)
- Performance: Good
- Implementation: Automatic (BigQuery 90+ days), or partition older data

Tier 3: Cold (Archive Storage)

- Data: 2+ years
- Access: Rarely (compliance, audits)
- Cost: Low ($2-5/TB/month)
- Performance: Slower (acceptable for infrequent access)
- Implementation: S3/GCS + external tables

Tier 4: Glacier (Deep Archive)

- Data: 5+ years
- Access: Almost never
- Cost: Lowest ($1/TB/month)
- Performance: Slow retrieval (hours)
- Implementation: S3 Glacier, GCS Archive

Implementation Example

Impact:

Cost Monitoring and Alerting

Essential Metrics to Track

1. Daily/weekly cost trends

```
-- Snowflake: Daily credit consumption
SELECT
DATE(start_time) as usage_date,
warehouse_name,
SUM(credits_used) as total_credits,
SUM(credits_used) * 3 as estimated_cost_usd -- Adjust based on your rate
FROM snowflake.account_usage.warehouse_metering_history
WHERE start_time >= DATEADD(day, -30, CURRENT_TIMESTAMP())
GROUP BY usage_date, warehouse_name
ORDER BY usage_date DESC, total_credits DESC;

-- BigQuery: Daily bytes processed
SELECT
DATE(creation_time) as usage_date,
user_email,
SUM(total_bytes_processed) / POW(10, 12) as tb_processed,
SUM(total_bytes_processed) / POW(10, 12) * 5 as estimated_cost_usd
FROM `region-us`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE creation_time >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 30 DAY)
AND job_type = 'QUERY'
AND state = 'DONE'
GROUP BY usage_date, user_email
ORDER BY usage_date DESC, tb_processed DESC;
```

2. Cost by user/team

3. Most expensive queries

```
-- BigQuery: Top 100 most expensive queries
SELECT
user_email,
query,
total_bytes_processed / POW(10, 12) as tb_processed,
total_bytes_processed / POW(10, 12) * 5 as cost_usd,
TIMESTAMP_DIFF(end_time, start_time, SECOND) as duration_seconds
FROM `region-us`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE creation_time >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 7 DAY)
AND job_type = 'QUERY'
AND state = 'DONE'
ORDER BY total_bytes_processed DESC
LIMIT 100;
```

4. Warehouse utilization

```
-- Snowflake: Warehouse idle time
SELECT
warehouse_name,
SUM(credits_used) as total_credits,
SUM(CASE WHEN query_count = 0 THEN credits_used ELSE 0 END) as idle_credits,
(idle_credits / total_credits * 100) as pct_idle
FROM snowflake.account_usage.warehouse_metering_history
WHERE start_time >= DATEADD(day, -7, CURRENT_TIMESTAMP())
GROUP BY warehouse_name
HAVING pct_idle > 10 -- Alert if >10% idle time
ORDER BY idle_credits DESC;
```

Set Up Alerts

Snowflake: Resource monitors

```
-- Daily budget alert
CREATE RESOURCE MONITOR daily_budget WITH
CREDIT_QUOTA = 300
FREQUENCY = DAILY
START_TIMESTAMP = IMMEDIATELY
TRIGGERS
ON 80 PERCENT DO NOTIFY
ON 100 PERCENT DO SUSPEND;

-- Email notifications configured in web UI
```

BigQuery: Budget alerts (GCP Console)

Redshift: CloudWatch alarms

```
# AWS CLI: Create alarm for high CPU
aws cloudwatch put-metric-alarm \
--alarm-name redshift-high-cpu \
--alarm-description "Alert when Redshift CPU > 80%" \
--metric-name CPUUtilization \
--namespace AWS/Redshift \
--statistic Average \
--period 300 \
--threshold 80 \
--comparison-operator GreaterThanThreshold \
--evaluation-periods 2 \
--alarm-actions arn:aws:sns:us-east-1:123456789:redshift-alerts
```

ROI Tracking Framework

Calculate Current Baseline

Step 1: Gather 30-day costs

```
Platform: Snowflake
Current monthly cost: $50,000

Breakdown:
- Compute: $32,000 (64%)
- Storage: $12,000 (24%)
- Data transfer: $4,000 (8%)
- Cloud services: $2,000 (4%)
```

Step 2: Identify optimization opportunities

```
Target optimizations:
1. Auto-suspend (5 warehouses always-on): $8,000/month potential savings
2. Right-size (3 oversized warehouses): $4,500/month potential savings
3. Query optimization (top 20 queries): $3,000/month potential savings
4. Storage tiering (archive 25TB): $5,000/month potential savings
5. Clustering (3 large tables): $2,500/month potential savings

Total potential: $23,000/month (46% reduction)
```

Step 3: Implement and measure

```
Week 1-2: Auto-suspend and right-sizing
Actual savings: $10,200/month (82% of potential)

Week 3-4: Query optimization
Actual savings: $2,400/month (80% of potential)

Month 2: Storage tiering
```

```
Actual savings: $4,200/month (84% of potential)

Month 3: Clustering
Actual savings: $1,800/month (72% of potential)

Total actual savings: $18,600/month (37% reduction)
ROI: 81% of potential achieved
```

Ongoing Tracking Dashboard

Key metrics:

```
1. Month-over-month cost trend
- Current: $31,400
- Previous: $50,000
- Change: -37%

2. Cost per query
- Current: $0.15
- Previous: $0.28
- Change: -46%

3. Cost per TB stored
- Current: $18.50 (vs $23 list price)
- Compression + lifecycle: 20% better than list

4. Warehouse utilization
- Average: 68% (good)
- Target: 60-80%

5. Cache hit rate
- Current: 28%
- Target: >25%
```

Common Cost Pitfalls

Pitfall 1: Always-On Dev/Test Environments

Problem:

```
5 development warehouses running 24/7
Cost: 5 × Medium × 730 hours × $6/hour = $21,900/month
Actual usage: ~10 hours/day, 5 days/week = ~220 hours/month
```

Solution:

```
Auto-suspend + schedule
Cost: 5 × Medium × 220 hours × $6/hour = $6,600/month
Savings: $15,300/month (70%)
```

Pitfall 2: No Query Timeout

Problem:

```
Runaway query scans entire 50TB table
Duration: 6 hours
Cost: X-Large × 6 hours = 48 credits = $144

Happens 2-3 times/month = $400+/month wasted
```

Solution:

```
-- Set query timeout
ALTER WAREHOUSE ANALYTICS_WH SET STATEMENT_TIMEOUT_IN_SECONDS = 3600; -- 1 hour max

-- Or user-level:
ALTER USER data_analyst SET STATEMENT_TIMEOUT_IN_SECONDS = 1800; -- 30 min max
```

Pitfall 3: Forgetting to Drop Unused Resources

Problem:

```
- 15 warehouses created over time, 8 no longer used
- 50 tables from old POCs, no longer accessed
- 200TB of test data from 2 years ago
```

Solution:

```
-- Audit unused warehouses
SELECT
warehouse_name,
MAX(start_time) as last_used
FROM snowflake.account_usage.query_history
GROUP BY warehouse_name
HAVING last_used < DATEADD(day, -90, CURRENT_TIMESTAMP());

-- Drop unused warehouses
DROP WAREHOUSE old_poc_warehouse;

-- Audit unused tables
SELECT
table_catalog,
table_schema,
table_name,
bytes,
MAX(last_altered) as last_modified
FROM snowflake.account_usage.tables
GROUP BY table_catalog, table_schema, table_name, bytes
HAVING last_modified < DATEADD(day, -180, CURRENT_TIMESTAMP())
AND bytes > 1000000000; -- >1GB
```

```
-- Archive or drop
DROP TABLE old_poc_data;
```

Pitfall 4: No Cost Ownership

Problem:

```
No one accountable for costs
Teams spin up large warehouses freely
No visibility into who's spending what
```

Solution:

```
1. Implement tagging/labeling:
- Tag warehouses with team name
- Use naming conventions: TEAM_PURPOSE_ENV

2. Create cost dashboards by team

3. Monthly cost reviews with each team

4. Set team budgets with resource monitors

5. Chargeback model (if applicable)
```

Implementation Roadmap

Implementation Roadmap

Phase 1: Quick Wins (Week 1-2)

Effort: Low | Savings: 20-35% | Risk: Low

- [ ] Enable auto-suspend on all warehouses (5-10 min idle)
- [ ] Right-size obviously oversized warehouses
- [ ] Pause non-production environments outside business hours
- [ ] Enable query result caching
- [ ] Set query timeouts
- [ ] Create resource monitors with alerts

Expected savings: $10,000-15,000/month on $50k baseline

Phase 2: Storage Optimization (Week 3-4)

Effort: Low-Medium | Savings: 15-25% | Risk: Low

- [ ] Identify and drop unused tables/databases
- [ ] Archive cold data (2+ years) to S3/GCS
- [ ] Reduce time travel retention where appropriate
- [ ] Implement transient tables for temporary data
- [ ] Enable automatic long-term storage (BigQuery)

Expected savings: $5,000-10,000/month

Phase 3: Query Optimization (Month 2)

Effort: Medium | Savings: 10-25% | Risk: Medium

- [ ] Identify top 20 most expensive queries
- [ ] Add partitioning/clustering to large tables
- [ ] Optimize query patterns (avoid full scans)
- [ ] Create materialized views for common aggregations
- [ ] Implement incremental processing

Expected savings: $5,000-12,000/month

Phase 4: Advanced Strategies (Month 3+)

Effort: Medium-High | Savings: 10-20% | Risk: Medium

- [ ] Implement multi-cluster warehouses
- [ ] Set up workload management
- [ ] Migrate appropriate workloads to serverless
- [ ] Implement data lifecycle policies
- [ ] Evaluate flat-rate pricing (BigQuery)
- [ ] Optimize distribution/sort keys (Redshift)

Expected savings: $5,000-10,000/month

Phase 5: Continuous Optimization (Ongoing)

Effort: Low (automated) | Savings: Maintained

- [ ] Weekly cost review meetings
- [ ] Monthly query performance audits
- [ ] Quarterly index/clustering reviews
- [ ] Automated alerts for anomalies
- [ ] Regular right-sizing assessments

Case Studies

Case Study Results: Monthly Savings Achieved

E-commerce	SaaS Analytics	Financial Svcs
$85K→$25K	$45K→$5.6K	$62K→$14K
71% saved	**88% saved**	**77% saved**

Case Study 1: E-commerce Company

Profile:

- Platform: Snowflake
- Initial cost: $85,000/month
- Data: 150TB
- Workload: ETL, reporting, ML

Optimizations:

1. **Auto-suspend (Week 1)**: 8 always-on warehouses → auto-suspend 5 min
   - Savings: $18,000/month (21%)
2. **Multi-cluster (Week 2)**: Production warehouse from 4X-Large static to Large multi-cluster (1-4)
   - Savings: $12,000/month (14%)
3. **Clustering (Month 2)**: Added clustering to 5 largest tables
   - Savings: $8,500/month (10%)
4. **Storage tiering (Month 2)**: Archived 80TB to S3
   - Savings: $15,000/month (18%)
5. **Query optimization (Month 3)**: Optimized top 30 queries
   - Savings: $6,500/month (8%)

Results:

- New monthly cost: $25,000
- Total savings: $60,000/month (71%)
- Implementation time: 3 months
- Annual savings: $720,000

Case Study 2: SaaS Analytics Platform

Profile:

- Platform: BigQuery
- Initial cost: $45,000/month
- Data: 200TB
- Workload: Customer dashboards, ad-hoc queries

Optimizations:

1. **Partitioning (Week 1)**: Partitioned 12 largest tables by date
    - Savings: $18,000/month (40%)
2. **Clustering (Week 2)**: Added clustering on customer_id
    - Savings: $6,500/month (14%)
3. **Materialized views (Month 2)**: Created 15 MVs for dashboard queries
    - Savings: $8,000/month (18%)
4. **Flat-rate pricing (Month 3)**: Switched from on-demand to 300 slots
    - Savings: $4,500/month (10%)
5. **Storage optimization (Month 3)**: Archived 120TB to GCS
    - Savings: $2,400/month (5%)

Results:

- New monthly cost: $5,600
- Total savings: $39,400/month (88%)
- Implementation time: 3 months
- Annual savings: $472,800

Case Study 3: Financial Services

Profile:

- Platform: Redshift
- Initial cost: $62,000/month
- Data: 80TB
- Workload: Regulatory reporting, analytics

Optimizations:

1. **Right-sizing (Week 1)**: 12-node RA3.4xlarge → 6-node
    - Savings: $19,000/month (31%)
2. **Pause/resume (Week 1)**: Dev clusters paused outside business hours
    - Savings: $8,500/month (14%)
3. **Workload management (Month 2)**: Configured WLM queues
    - Performance improvement: 40% faster queries
    - Enabled further downsizing: $4,000/month (6%)
4. **Spectrum (Month 2)**: Moved 50TB archive data to S3
    - Savings: $10,500/month (17%)
5. **Concurrency scaling (Month 3)**: Used for peak load instead of over-provisioning
    - Savings: $6,000/month (10%)

Results:

- New monthly cost: $14,000
- Total savings: $48,000/month (77%)

- Implementation time: 3 months
- Annual savings: $576,000

Appendix

Cost Calculator Templates

Snowflake cost calculator:

```
Monthly cost = (Warehouse credits × Hours × Credit price) + Storage + Transfer

Example:
- Warehouses:
- Production (Large): 4 credits/hr × 300 hrs × $3 = $3,600
- ETL (X-Large): 8 credits/hr × 100 hrs × $3 = $2,400
- Analytics (Medium): 2 credits/hr × 400 hrs × $3 = $2,400
- Storage: 25TB × $23/TB = $575
- Transfer: 500GB × $0.09/GB = $45

Total: $9,020/month
```

BigQuery cost calculator:

```
On-demand:
Monthly cost = TB processed × $5/TB

Example: 800TB/month = $4,000

Flat-rate:
Monthly cost = Slots × $20/slot

Example: 300 slots = $6,000

Choose flat-rate if processing > 1,200TB/month
```

Glossary

- **Auto-suspend**: Automatically pause warehouse after period of inactivity
- **Auto-resume**: Automatically start warehouse when query submitted
- **Clustering**: Physical organization of data to improve query performance
- **Credits**: Snowflake billing unit (1 credit = 1 hour of Small warehouse)
- **DWU**: Data Warehouse Units (Azure Synapse compute measure)
- **Materialized view**: Pre-computed query results stored as table
- **Multi-cluster**: Multiple warehouse instances for handling concurrency
- **Partitioning**: Dividing table into segments based on column value
- **Result caching**: Reusing query results when same query run multiple times
- **Slots**: BigQuery compute capacity units
- **Spectrum**: Redshift feature for querying data in S3

Resources

- **Snowflake Cost Management**: https://docs.snowflake.com/en/user-guide/cost-understanding
- **BigQuery Pricing**: https://cloud.google.com/bigquery/pricing
- **Redshift Best Practices**: https://docs.aws.amazon.com/redshift/latest/dg/best-practices.html
- **Azure Synapse Optimization**: https://docs.microsoft.com/azure/synapse-analytics/sql-data-warehouse/

Document Version: 1.0

Last Updated: 2025

For questions or feedback, visit: https://dataco.com/resources

www.EnterprisedDataSolutions.co.nz | Contact@EnterprisedDataSolutions.co.nz