Serverless Data Processing & Cloud-Native Query Engines

Part 2

Cloud-Native Query Engines

- serverless DB engines that enable SQL-based querying directly against data stored in cloud storage systems.

Key Characteristics:

- **Separation of Storage and Compute**: Independent scaling and management
- **Serverless or Auto-scaling**: Resources allocated on demand
- **SQL Interface**: Standard query language for data access
- **Schema-on-Read**: Apply schema at query time
- **Pay-per-Query**: Billing based on data scanned or compute used
- **No Infrastructure Management**: Fully managed service
- **Multi-format Support**: Query various file formats (Parquet, ORC, JSON, CSV)

Benefits Over Traditional Databases

- **Cost Efficiency**: Pay only for queries executed
- **Scalability**: Automatic scaling to match query complexity
- **Simplified Operations**: No cluster management
- **Flexibility**: Query data in various formats and locations
- **Integration**: Native integration with data lake storage
- **Accessibility**: SQL interface for broad user adoption
- **Performance**: Optimized for analytical workloads

Traditional vs. Serverless Data Processing

Traditional Processing

Infrastructure

- Fixed infrastructure (on-premises or cloud VMs)
- Dedicated clusters (Hadoop, Spark, etc.)
- Manual scaling procedures
- Capacity planning required
- High availability configuration

Operational Aspects

- Cluster management overhead
- Software installation and updates
- Monitoring and alerting setup
- Backup and recovery procedures
- Dedicated operations team

Serverless Processing

Infrastructure

- No infrastructure management
- Automatic scaling based on workload
- No capacity planning
- Built-in high availability
- Event-driven architecture

Operational Aspects

- Minimal operational overhead
- Automatic updates and patches
- Built-in monitoring
- Managed backup and recovery
- Reduced operations team requirements

Traditional vs. Serverless Data Processing

Traditional Processing

Development Experience

- Infrastructure-aware development
- Configuration management
- Resource allocation considerations
- Performance tuning

Cost Model

- Capital expenditure or fixed costs
- Pay for allocated resources
- Continuous resource utilization
- Overprovisioning for peak loads
- Complex licensing models

Serverless Processing

Development Experience

- Focus on business logic
- Simplified deployment
- Configuration as code
- Event-driven programming model
- Reduced performance tuning

Cost Model

- Pay only for actual usage
- Zero cost when idle
- Granular billing (per invocation, per second)
- No upfront costs
- Predictable pricing for specific workloads

Traditional vs. Serverless Data Processing

Aspect	Traditional	Serverless
Infrastructure Management	User responsibility	Provider managed
Scaling	Manual or auto-scaling	Automatic, including to zero
Cost Model	Pay for allocation	Pay for usage
Idle Costs	Yes	No
Cold Start	No	Yes
Operational Overhead	High	Low
Development Focus	Infrastructure + Logic	Business Logic
Maximum Runtime	Unlimited	Limited (minutes to hours)
State Management	Built-in	External services required
Vendor Lock-in	Lower	Higher

When to Use Serverless Data Processing

- Variable Workloads
- Event-Driven Processing
- Microservices Architecture
- Rapid Development
- Cost Optimization

Considerations and Limitations

- Cold Start Latency
- Execution Time Limits
- State Management
- Vendor Lock-in
- Debugging Complexity
- Cost Predictability

Major Cloud-Native Query Engines: Amazon Athena

- Serverless interactive query service
- Built on Presto and Apache Hive
- Query data directly in Amazon S3
- Standard SQL support (ANSI SQL)
- Pay-per-query pricing ($5 per TB scanned)
- No infrastructure to manage

•Launched in 2016

Athena Use Cases

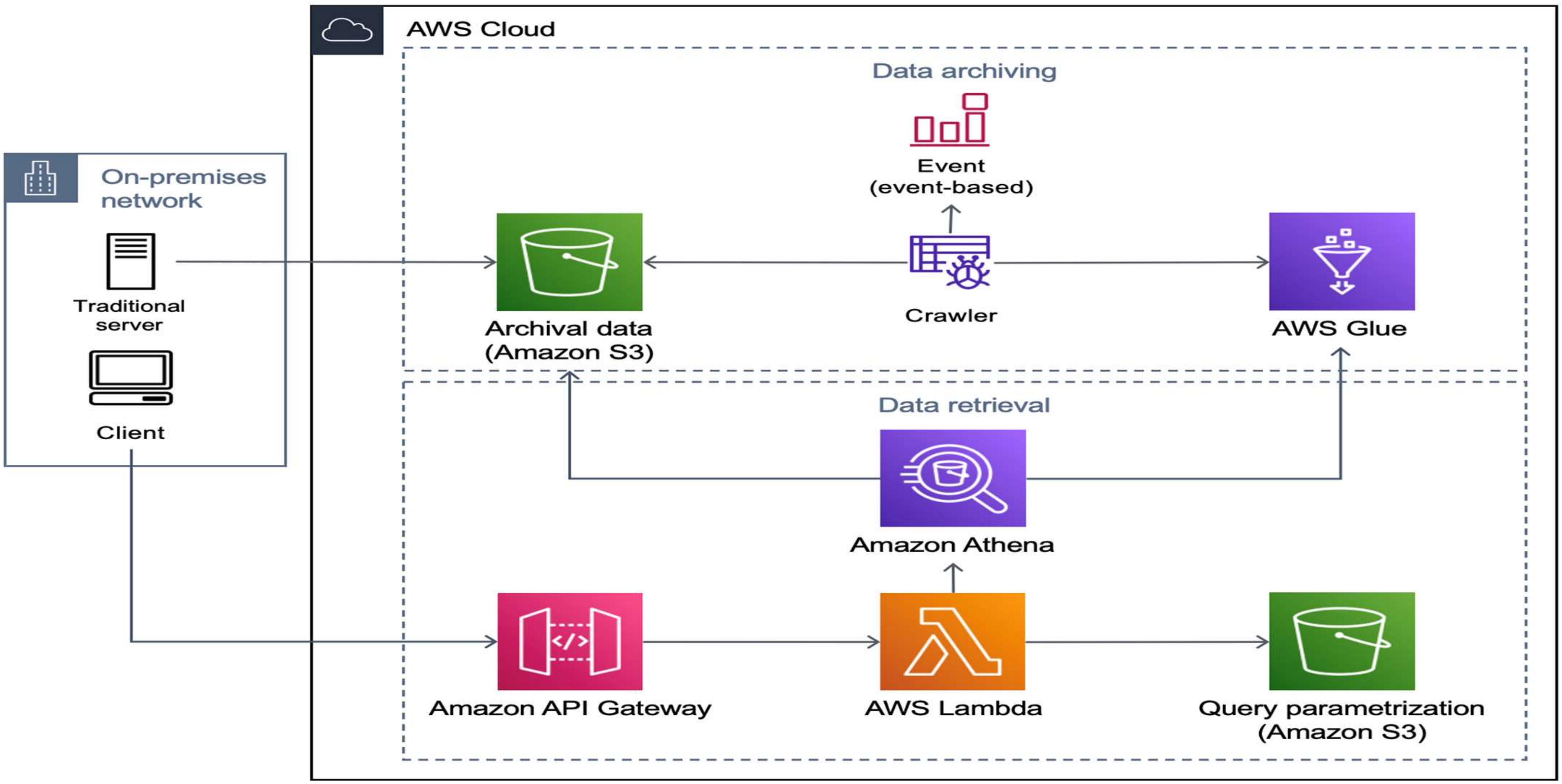
- **Ad-hoc Analysis**: Interactive exploration of data lake
- **Business Intelligence**: Power dashboards and reports
- **Data Preparation**: Query and transform data for further processing
- **Log Analysis**: Query and analyze application logs
- **Data Science Preparation**: Explore and prepare data for ML
- **Federated Queries**: Query across multiple data sources

```sql
-- Create a table with partitioning
CREATE EXTERNAL TABLE sales (
    transaction_id string,
    customer_id string,
    product_id string,
    quantity int,
    price decimal(10,2)
)
PARTITIONED BY (year int, month int, day int)
STORED AS PARQUET
LOCATION 's3://my-bucket/sales/';

-- Add partitions
ALTER TABLE sales ADD
PARTITION (year=2023, month=3, day=1) LOCATION 's3://my-bucket/sales/year=2023/month=3/day=1/'
PARTITION (year=2023, month=3, day=2) LOCATION 's3://my-bucket/sales/year=2023/month=3/day=2/';

-- Query with partition pruning
SELECT customer_id, SUM(price * quantity) as total_spend
FROM sales
WHERE year = 2023 AND month = 3 AND day = 1
GROUP BY customer_id
ORDER BY total_spend DESC
LIMIT 10;
```

Amazon Athena Architecture



Source: https://aws.amazon.com/blogs/architecture/reduce-archive-cost-with-serverless-data-archiving/

Major Cloud-Native Query Engines: Google BigQuery

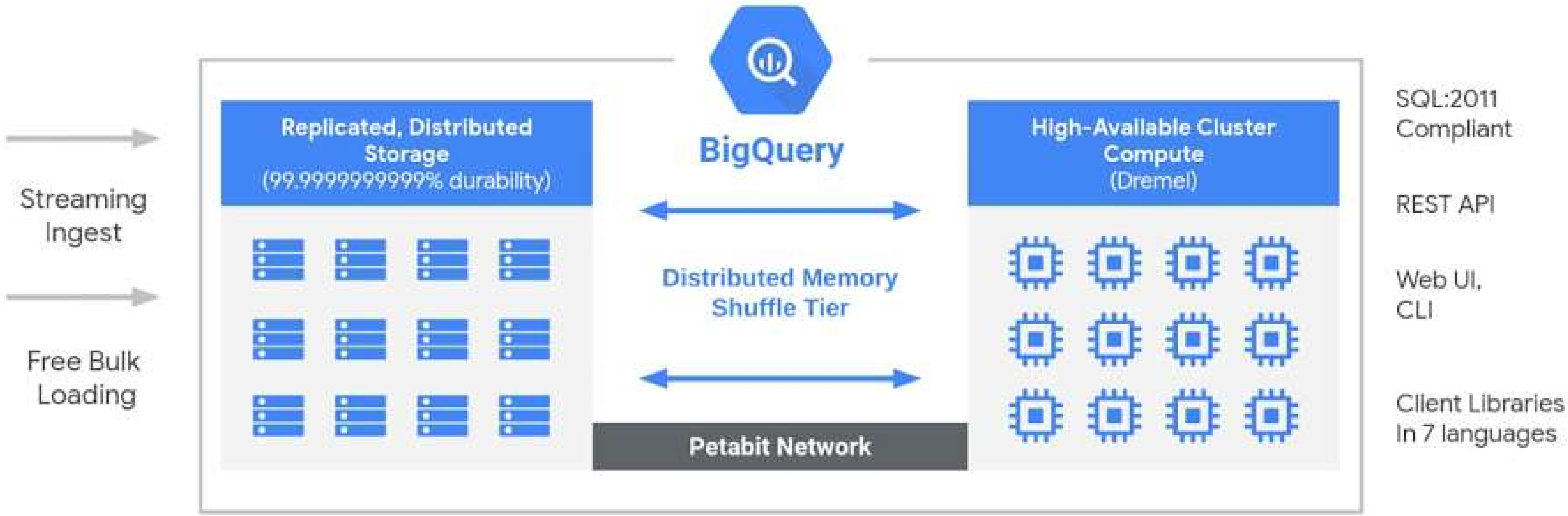
- Fully managed, serverless data warehouse
- Separates storage and compute
- Massive parallel processing architecture
- Pay-per-query pricing with flat-rate options
- Automatic scaling and high availability
- Built-in machine learning capabilities
- Launched in 2010

BigQuery Use Cases

- **Enterprise Data Warehousing**: Centralized analytics platform
- **Real-time Analytics**: Streaming data analysis
- **Machine Learning**: Integrated ML capabilities
- **Data Sharing**: Secure cross-organization sharing
- **IoT Analytics**: Process and analyze device data
- **Log Analysis**: Analyze application and system logs

```sql
-- Create a table
CREATE OR REPLACE TABLE retail_dataset.sales (
  transaction_id STRING,
  customer_id STRING,
  product_id STRING,
  quantity INT64,
  price NUMERIC,
  transaction_date DATE
);

-- Complex analytical query with window functions
SELECT
  customer_id,
  transaction_date,
  price * quantity AS purchase_amount,
  SUM(price * quantity) OVER (
    PARTITION BY customer_id
    ORDER BY transaction_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS cumulative_spend,
  RANK() OVER (
    PARTITION BY customer_id
    ORDER BY price * quantity DESC
  ) AS purchase_rank
FROM retail_dataset.sales
WHERE transaction_date BETWEEN '2023-01-01' AND '2023-03-31'
ORDER BY customer_id, transaction_date;
```



BigQuery Architecture

Source: https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview

Major Cloud-Native Query Engines: SnowFlake

- Cloud-native data platform
- Multi-cloud support (AWS, Azure, GCP)
- Separation of storage, compute, and services
- Virtual warehouses for compute resources
- Automatic scaling and concurrency
- Time travel and zero-copy cloning
- Founded in 2012, public in 2020

Snowflake Use Cases

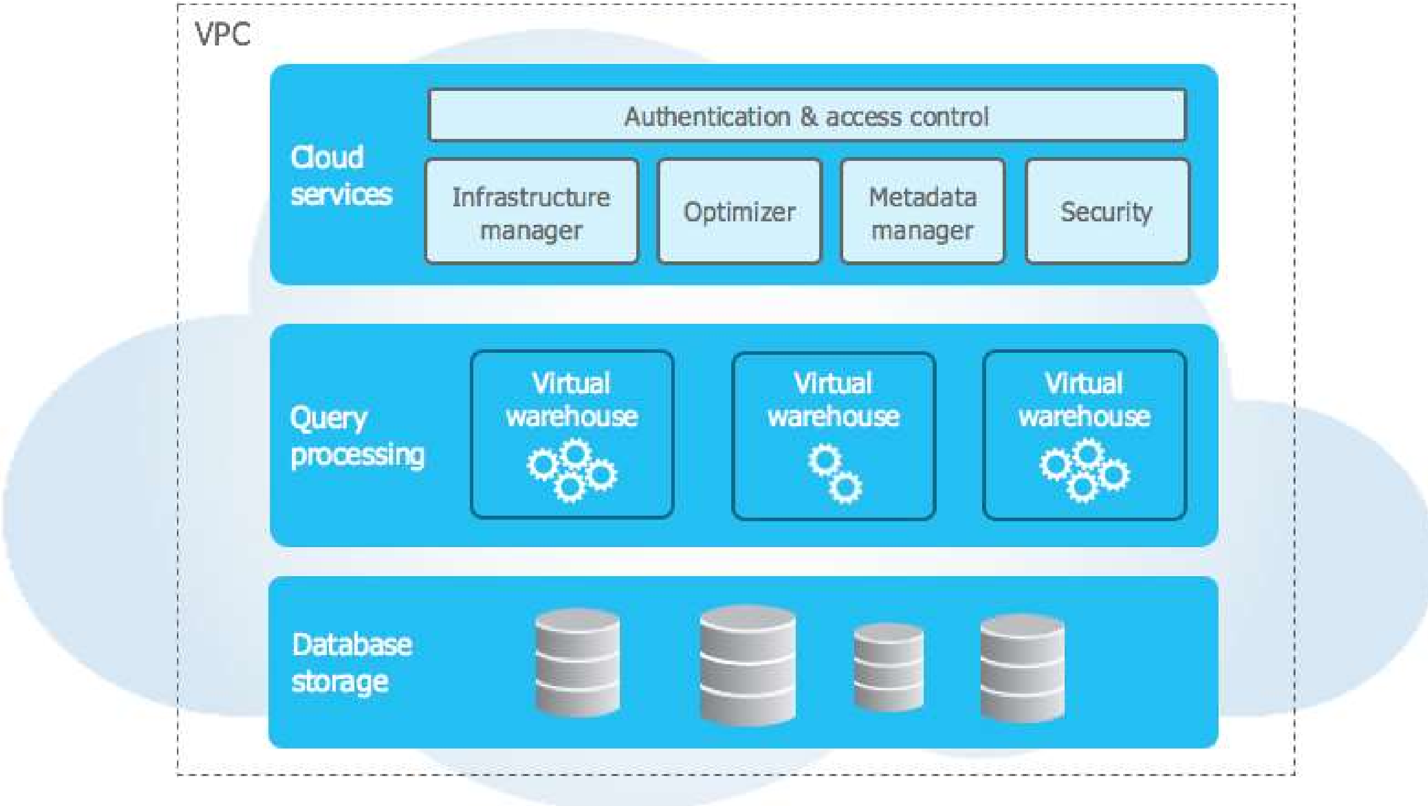
- **Data Warehousing**: Enterprise analytics platform
- **Data Lake Integration**: Query and process data lake data
- **Data Applications**: Power data-intensive applications
- **Data Sharing**: Secure data exchange ecosystem
- **Data Engineering**: ETL/ELT pipelines
- **Data Science**: Prepare and analyze data for ML

```sql
-- Create a table with VARIANT type for JSON
CREATE OR REPLACE TABLE customer_events (
  event_id VARCHAR,
  event_timestamp TIMESTAMP_NTZ,
  event_data VARIANT
);

-- Insert JSON data
INSERT INTO customer_events
SELECT
  '12345',
  CURRENT_TIMESTAMP(),
  PARSE_JSON('{"customer_id": "C123", "event_type": "purchase", "items": [{"product_id": "P1",

-- Query JSON data using dot notation and FLATTEN
SELECT
  event_id,
  event_timestamp,
  event_data:customer_id::STRING AS customer_id,
  event_data:event_type::STRING AS event_type,
  item.value:product_id::STRING AS product_id,
  item.value:quantity::INT AS quantity,
  item.value:price::DECIMAL(10,2) AS price
FROM customer_events,
LATERAL FLATTEN(input => event_data:items) AS item;
```

SnowFlake Architecture



Source: https://docs.snowflake.com/en/user-guide/intro-key-concepts

Participation Question

A data analytics team needs to build a new reporting system that will query data from multiple sources (S3 data lake, operational databases, and third-party APIs). They need to generate both scheduled reports and support ad-hoc analysis by business users. The workload is highly variable, with end-of-month processing requiring 10x the normal capacity.

Q: How would you design the architecture to efficiently handle the variable workload while maintaining good performance for scheduled and ad-hoc queries?

Real-World Applications and Case Studies

Netflix: BigQuery for Content Analytics

Challenge

- Petabytes of viewing data
- Need for real-time content performance insights
- Complex analytical queries
- Global scale requirements
- Integration with data science workflows

Solution

- BigQuery as central analytics platform
- Streaming ingest for real-time data
- Custom data pipelines for transformation
- Integration with visualization tools
- ML models for recommendation system

Results

- Real-time content performance analytics
- Personalization algorithm training
- Cost optimization through query tuning
- Improved decision-making speed
- Integration with internal data science tools

Airbnb: AWS Lambda for Real-time Data Processing

Challenge

- Billions of events daily
- Need for real-time processing
- Variable workload patterns
- Complex business logic
- Global distribution requirements

Solution

- AWS Lambda for event processing
- Kinesis for data streaming
- DynamoDB for state management
- Step Functions for orchestration
- CloudWatch for monitoring

Results

- Processes billions of events daily
- Real-time fraud detection
- Dynamic pricing calculations
- Personalized search rankings
- Event-driven architecture

Capital One: Serverless Data Processing Pipeline

Challenge

- Legacy Hadoop infrastructure
- High operational costs
- Long development cycles
- Difficulty scaling for peak loads
- Complex data transformation needs

Solution

- Migrated from Hadoop to serverless
- AWS Lambda for data transformation
- Amazon Athena for ad-hoc analysis
- Step Functions for orchestration
- S3 for data lake storage

Results

- Reduced operational costs by 60%
- Improved development velocity
- Better handling of variable workloads
- Enhanced data quality
- Simplified architecture

Summary

- Cloud-native query engines enable SQL-based querying directly on cloud storage
- Major providers offer robust solutions (Amazon Athena, Google BigQuery, Snowflake) with different strengths
- Separation of storage and compute optimizes cost and performance
- Serverless data processing eliminates infrastructure management and provides true pay-per-use pricing
- Ideal use cases include variable workloads, event-driven processing, and cost-sensitive scenarios