Time-Series Databases

Group A4: George Vlady, John Wright Stanly, Spencer Pierce Kessler, Yulu Liu

Intro: Definition & Key Concepts

• Time Series Databases (TSDBs) are specialized database systems designed to store, retrieve, and process time-stamped data points collected at regular intervals.

• Key terms and concepts:

- Time Series: Sequential data points indexed in time order
- Data Point: A measurement or event with a timestamp
- Retention Policy: Rules for how long data is stored
- Downsampling: Reducing data resolution over time

• Emerged in early 2000s as traditional databases struggled with the volume and velocity of time-based data

Intro: Why Time Series Databases Matter

- Primary purpose: Efficiently manage and analyze large volumes of time-stamped data at scale

- Business value:
  - Real-time monitoring and alerting
  - Performance optimization through historical analysis
  - Predictive maintenance and forecasting
  - Regulatory compliance through historical records

- Transforming industries by enabling data-driven decisions based on temporal patterns

- Current market size: $320.6 million in 2022, projected to reach $711.3 million by 2030 with a CAGR of 10.68%

(https://www.kingsresearch.com/time-series-databases-software-market-110)

Product Overview-features

- **High Ingestion Rates**: Optimized for fast data writes, handling millions of events per second.
- **Efficient Storage**: Uses compression and data retention policies to optimize storage costs.
- **Indexing & Query Performance**: Provides efficient indexing mechanisms for fast queries on large datasets.
- **Scalability & Distributed Architecture**: Supports horizontal scaling for large-scale applications.
- **Retention Policies & Downsampling**: Enables automated data summarization to balance granularity and storage.
- **Integration with Analytics & Visualization**: Seamless connection with Grafana, Prometheus, and other analytics tools.
- **Streaming & Real-Time Processing**: Supports real-time analytics with event-driven processing.

Feature	InfluxDB	TimescaleDB	OpenTSDB
High Ingestion Rate	✅	✅	✅
Compression & Storage Efficiency	✅	✅	✅
SQL Query Support	❌	✅	❌
Scalability	✅	✅	✅
Indexing Optimization	✅	✅	✅
Retention & Downsampling	✅	✅	❌
Integration with Grafana	✅	✅	✅
Real-Time Processing	✅	✅	✅

More comparison on Query language and ecosystem

Feature	**InfluxDB**	**TimescaleDB**	**OpenTSDB**
Query Language	InfluxQL, Flux	SQL (PostgreSQL)	OpenTSDB Query API
Joins & Aggregations	Limited joins, powerful aggregations in Flux	Supports joins, complex analytics	Limited support for joins
Data Visualization	Integrates with Grafana, Chronograf	Works with Grafana, Tableau, PostgreSQL clients	Works with Grafana, OpenTSDB UI

Overall

Feature	**InfluxDB**	**TimescaleDB**	**OpenTSDB**
Best Use Cases	IoT, DevOps monitoring, real-time analytics	Financial time-series, business analytics	Large-scale infrastructure monitoring
Pros	Fast ingestion, purpose-built for time-series, rich ecosystem	SQL support, seamless PostgreSQL integration, good compression	Scales well for massive datasets, HBase reliability
Cons	No native SQL, clustering is enterprise-only	Overhead of PostgreSQL, slower ingestion than InfluxDB	HBase dependency, query latency issues

Research Comparison

Feature	TARDIS	BTrDB	Gorilla
High-Resolution Storage	✅	✅	✅
High-Speed Queries	✅	✅	✅
Compression Mechanism	✅	✅	✅
Designed for Distributed Systems	✅	✅	❌
Efficient Data Summarization	✅	✅	❌
Open Source	❌	✅	✅

Overall

Feature	**TARDIS**	**BTrDB**	**Gorilla**
Best Use Cases	Scientific modeling, physics, astronomy	Real-time power grid & IoT analytics	Large-scale system monitoring at Facebook
Pros	Optimized for scientific data, precise simulations	Ultra-low latency, high throughput	Efficient in-memory compression
Cons	Niche use case, not general-purpose	Requires distributed setup for full scalability	Not available outside Facebook

Technical Detail - GorillaDB

Purpose: Designed for real-time monitoring of Meta's infrastructure and web services.

Optimized for:

- High ingestion rates: Handles millions of metrics per second.
- Recent data queries: 85% of queries target the last 26 hours.
- Low-latency analytics: Enables fast aggregation over small windows (e.g., 10s intervals).

Primary Use Cases:

- Infrastructure & application monitoring.
- Real-time anomaly detection and alerting.
- Performance telemetry for large-scale distributed systems.

Key Trade-Offs:

- Prioritizes speed and efficiency over data flexibility.
- Limited schema: String key, timestamp, double-precision float.
- Description

Gorilla DB Architecture & Innovations

Memory-First Design: Stores recent data (~24 hours) entirely in-memory, ensuring ultra-fast reads.

Time-Series Compression:

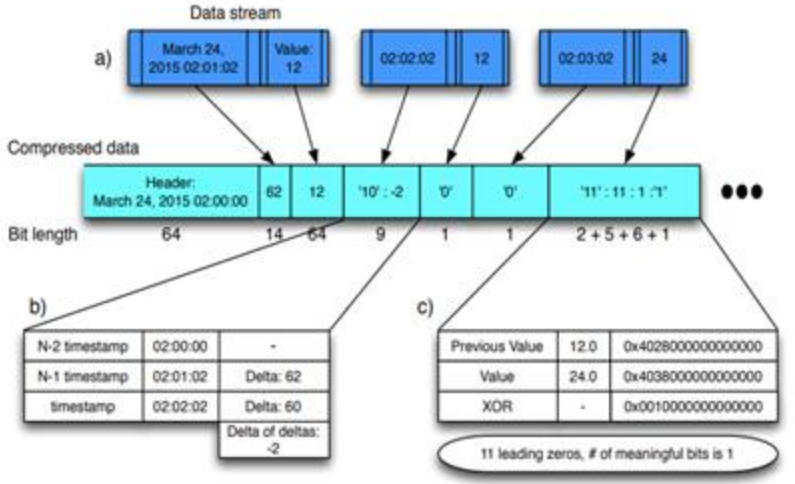
- Uses XOR-based delta encoding to store timestamps efficiently.
- Floating-point values are compressed with a specialized XOR-based algorithm, reducing space usage significantly.
- Compression achieves 10x reduction in storage needs while maintaining query speed.

Fault Tolerance & Availability:

- Replicated across multiple instances for resilience.
- Loss of individual data points does not impact aggregated results due to time-based rollups.

Read/Write Optimization:

- In-memory indexing for fast access.
- Write-through caching model—data eventually moves to a long-term storage backend for historical queries.



Technical Detail - TimescaleDB

Purpose: Extends PostgreSQL to efficiently handle large-scale time-series data.

Optimized for:

- High write throughput: Handles millions of inserts per second.
- Efficient historical queries: Optimized for long-range scans across large datasets.
- Seamless SQL integration: Enables advanced analytics using standard SQL queries.

Primary Use Cases:

- IoT telemetry (sensor networks, industrial data).
- Financial data analysis (stock prices, crypto markets).
- Server and infrastructure monitoring.

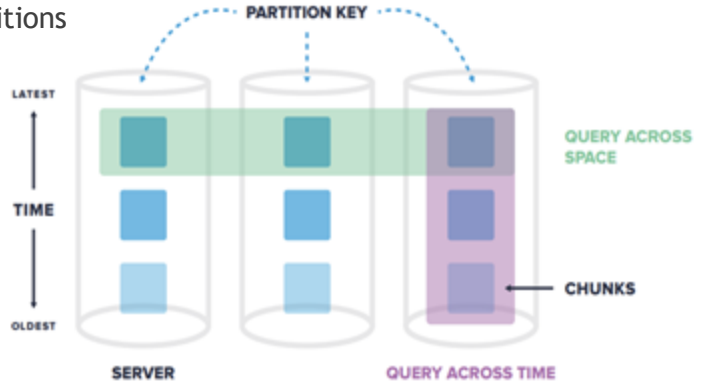
Key Trade-Offs:

- Built on relational DB principles (ie. ACID), which may not be as optimized as NoSQL TSDBs for ultra-low-latency workloads.
- Storage and query performance depend on partitioning strategy.

TimescaleDB Architecture & Innovations

Key Innovation - Hypertables & Chunking:

- Users interact with a single hypertable, while TimescaleDB automatically partitions data into smaller “chunks” based on time and an optional secondary key.
- Each chunk is a standard PostgreSQL table, enabling efficient indexing and querying.



Advanced Compression:

- Uses compression for numerical data (similar to Facebook’s Gorilla DB).
- Achieves 5-10x storage reduction while maintaining query performance.

Native SQL Support:

- Fully compatible with PostgreSQL, enabling complex queries using standard SQL.
- Supports window functions, JOINs, aggregations, and real-time analytics.

Scalability & Distributed Querying:

- Multi-node support allows horizontal scaling across clusters.
- Optimized query planner routes queries efficiently across partitions.

Case Study 1: Discover - InfluxDB

Company Overview: Discover Financial is a major U.S. financial services company known for the “Discover Card”

- Challenge: Needed an observability platform to monitor financial transactions and IT infrastructure
- Solution: Implemented InfluxDB Cloud for
  - Real-time monitoring of payments and services
  - Proactive issue detection using Grafana dashboards
  - Scalability and cloud-based insights
- Impact: Improved service uptime and customer experience reliability.

https://www.influxdata.com/customer/discover/

Case Study 2: WaterBridge - Timescale DB

Company Overview: Waterbridge is an oil and gas company that specializes in hydraulic fracturing water treatment and disposal

- Challenge: Needed real-time data consistency for operational monitoring and safety
- Solution: Implemented TimescaleDB for
  - Ingest 5,000–10,000 data points per second for real-time monitoring
  - Track water pressure, flow, and temperature in pipelines for leak detection and equipment management
  - Visualize real-time data using tools like PowerBI, Seeq, and Spotfire
- Impact: Improved operational safety and efficiency through real-time data tracking.

https://www.timescale.com/blog/how-waterbridge-uses-timescaledb-for-real-time-data-consistency

InfluxDB Demo

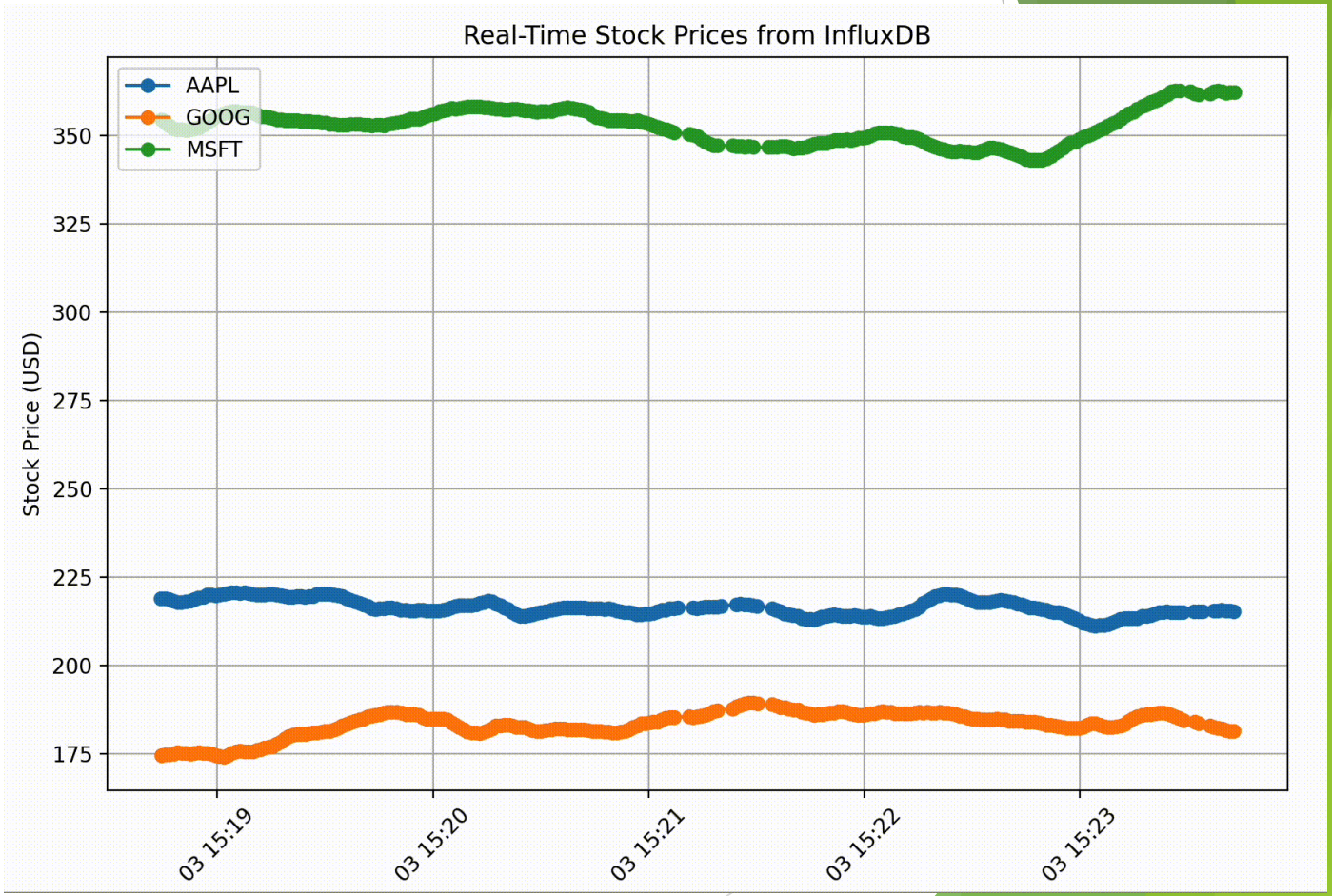
```python
import time
import numpy as np
from datetime import datetime
from influxdb_client import InfluxDBClient, Point, WritePrecision

# InfluxDB Cloud credentials
INFLUXDB_URL = "https://us-east-1-1.aws.cloud2.influxdata.com"
INFLUXDB_TOKEN = '
ORG = "CS4440_Demo"
BUCKET = "demo_bucket"

# Connect to InfluxDB
client = InfluxDBClient(url=INFLUXDB_URL, token=INFLUXDB_TOKEN, org=ORG)
from influxdb_client.client.write_api import SYNCHRONOUS

write_api = client.write_api(write_options=SYNCHRONOUS)

# Initialize stock prices
stock_prices = {"AAPL": 250.0, "GOOG": 175.0, "MSFT": 375.0}
previous_changes = {symbol: 0 for symbol in stock_prices}  # Store inertia for each stock
alpha = 0.8  # Smoothing factor (higher = more inertia)

def simulate_stock_price(symbol):

def write_stock_data():
    """ Generate and write stock prices to InfluxDB """
    while True:
        for stock in stock_prices:
            price = simulate_stock_price(stock)
            point = (
                Point("stock_price")
                .tag("symbol", stock)
                .field("price", price)
                .time(datetime.utcnow(), WritePrecision.NS)
            )
            write_api.write(bucket=BUCKET, org=ORG, record=point)
            print(f"Written: {stock} - ${price}")

        time.sleep(1)  # Wait 1 second before next update

if __name__ == "__main__":
    write_stock_data()
```

InfluxDB Demo

```python
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from influxdb_client import InfluxDBClient

# InfluxDB Cloud credentials
INFLUXDB_URL = "https://us-east-1-1.aws.cloud2.influxdata.com"
INFLUXDB_TOKEN = '
ORG = "CS4440_Demo"
BUCKET = "demo_bucket"

client = InfluxDBClient(url=INFLUXDB_URL, token=INFLUXDB_TOKEN, org=ORG)
query_api = client.query_api()

# Initialize the figure and axis for Matplotlib
fig, ax = plt.subplots(figsize=(10, 5))
lines = {}  # Dictionary to store line plots for each stock symbol
stock_symbols = ["AAPL", "GOOG", "MSFT"]  # Track multiple stocks

def fetch_stock_data(symbol):
    """ Query last 5 minutes of stock data """
    query = f'''
    from(bucket: "{BUCKET}")
      |> range(start: -5m)
      |> filter(fn: (r) => r["_measurement"] == "stock_price")
      |> filter(fn: (r) => r["symbol"] == "{symbol}")
      |> filter(fn: (r) => r["_field"] == "price")
      |> yield(name: "prices")
    '''
    tables = query_api.query(query, org=ORG)

    times, prices = [], []
    for table in tables:
        for record in table.records:
            times.append(record.get_time())
            prices.append(record.get_value())

    return times, prices

def update(frame):-

# Set up real-time animation (updates every 2 seconds)
ani = animation.FuncAnimation(fig, update, interval=2000)

# Show the real-time plot
plt.show()
```

InfluxDB Demo

```
Written: AAPL - $220.0
Written: GOOG - $160.27
Written: MSFT - $347.87
Written: AAPL - $220.22
Written: GOOG - $160.22
Written: MSFT - $348.22
Written: AAPL - $220.18
Written: GOOG - $160.16
Written: MSFT - $348.77
Written: AAPL - $219.8
Written: GOOG - $160.14
Written: MSFT - $349.61
Written: AAPL - $219.54
Written: GOOG - $160.07
Written: MSFT - $350.19
Written: AAPL - $219.23
Written: GOOG - $159.95
Written: MSFT - $350.97
Written: AAPL - $219.28
Written: GOOG - $159.77
Written: MSFT - $351.37
Written: AAPL - $219.35
Written: GOOG - $159.76
Written: MSFT - $351.84
Written: AAPL - $219.25
Written: GOOG - $159.86
Written: MSFT - $351.97
Written: AAPL - $219.15
Written: GOOG - $160.0
Written: MSFT - $352.07
Written: AAPL - $219.27
Written: GOOG - $160.33
Written: MSFT - $352.42
Written: AAPL - $219.36
Written: GOOG - $160.49
Written: MSFT - $352.92
Written: AAPL - $219.22
Written: GOOG - $160.87
Written: MSFT - $353.88
Written: AAPL - $219.15
Written: GOOG - $161.16
Written: MSFT - $354.74
```



Future Trends: How Time Series DBs are Evolving

- Recent developments:
  - Cloud-native architectures (since 2018)
  - Integration with stream processing frameworks (since 2020)
  - Enhanced machine learning capabilities (since 2021)

- Key drivers of change:
  - IoT explosion (billions of connected devices)
  - Edge computing requirements
  - Growing demand for real-time analytics
  - Increasing data volumes from monitoring systems

Future Trends: What to Expect in the Next 1-3 Years

- Emerging capabilities:
  - Automated anomaly detection and forecasting
  - Enhanced compression algorithms for storage efficiency
  - Serverless time series database offerings
  - Improved query languages for complex time-based analysis

- Industry adoption trends:
  - Wider adoption in manufacturing and industrial IoT
  - Integration with observability platforms
  - Shift from general-purpose databases to specialized TSDBs

- Potential disruptions: Convergence with streaming platforms

Future Trends: Long Term Landscape 3-5 Years

- Technology maturation:
  - Federated time series databases across cloud and edge
  - AI-driven automated database optimization
  - Quantum computing applications for complex time series analysis

- Potential breakthrough areas:
  - Real-time digital twins powered by time series data
  - Causal inference at scale for complex systems

- Integration with other technologies:
  - Blockchain for immutable audit trails of critical time series data
  - AR/VR visualizations of complex temporal patterns

Future Trends: Challenges and Opportunities

- Key challenges:
  - Balancing performance with storage costs
  - Managing data privacy across jurisdictions
  - Handling increasingly complex data relationships
  - Scaling to accommodate exponential data growth

- Emerging opportunities:
  - Specialized industry-specific TSDB solutions
  - Time series as a service (TSaaS) business models
  - Cross-domain analytics combining time series with other data types

- What to watch: Convergence of time series, graph, and spatial databases