Section 1: Week 2: Reproduce an Experiment

Nate Bachmeier

TIM-7020: Database and Business Intelligence

January 26th, 2020

North Central University

Exploring the Not Only SQL World

## Describe the Issue

Even trivial applications need to be cognizant of the differences between vectors, maps, and trees. If the application needs to perform millions of random accesses, then the vector would underperform the map, because of the lack of a key-index. The inverse is also true that for sequential reads, where the map will underperform as that is not its intended use case. These same concepts of access-pattern specific data structures apply to larger and more complex environments. As the data becomes larger and needs redundancy, availability, and persistence, these data structures need to become encapsulated into data management solutions. Just as there are multiple data structures with distinct scenario-specific optimizations, this concept recurses into the design decisions of different data management solutions. Not Only SQL (NoSQL) is a broad category of technologies that exploit these distinctions to enable built-for-purpose expert systems, in contrast to traditional general-purpose relational technologies. With the emergence of IoT, Cloud, Big Data, and Mobile (ICBM)—businesses need to adopt NoSQL solutions that are specific to the problem and not assume that one size fits all. When the business chooses the correct technology, then it becomes easier to gain insights by transforming their data into business intelligence. Organizations that are capable of making timely decisions are more productive and competitive in dynamic market conditions.

For instance, a traditional representation of a social network is a graph, and using graph databases (e.g., Apache TinkerPop and Neo4J) will likely require fewer resources than a general relational store (e.g., Azure SQL and Postgres). The primary reason for these gains comes from each node contains a local index of edges versus a relational store, which relies on global indexes that have more noise to filter. The same behavior occurs with time-series information that needs to perform sequential reads and tools like Influx and OpenTSDB focus on these specifics. Big data platforms like Azure Data Lake Analytics and Amazon Athena, attempt to negate the value of these specialized indexes by horizontally scaling partitioned data. While horizontal scaled compute can address most data management challenges, it can quickly become prohibitively expensive for smaller organizations (and doctoral students).

Martino et al. (2019) state that an official benchmark suite does not exist for time-series data management, and this introduces challenges for comparing different NoSQL solutions. To mitigate this limitation, they ingested real-world Industrial Internet of Things (IIoT) workloads into a document store (Mongo), a columnar store (Casandra), and a time-series store (Influx). The researcher then evaluated batch-ingestion times, retrieval, and disk usage of the different platforms. Balis et al. (2017) discuss a similar comparison test between a document store (Mongo), relational store (Postgres), in-memory cache (Redis), and a time-series store (Influx). They also relied on performance counters to measure memory, disk, and query times to perform random access fetches of 1,000 records. Anikin et al. (2019) also evaluate the resource utilization characteristics between various graph databases (e.g., Apache HBase, Apache TinkerPop) against a relational store (Postgres) and in-memory platform (Apache Spark).

## Describe the accomplishments and limitations

A reoccurring theme across these tests is that using an expert system outperforms a general-purpose solution. The authors collected statistical data to make comparisons between the different technologies; however, except for Balis et al., the queries are not provided. There is also little mentioned around technology-specific nuances. For instance, Amazon DynamoDB, a key-value store, can emulate fast sequential scans using the starts-with operator to retrieve multiple related items in a single query. When users do not follow these design-specific patterns, then the same operation could require multiple fetches and could be perceived as less optimal. Redis supports hosting in-memory vectors and maps, which enables developers to arbitrarily index data for different access-patterns. If there is a misalignment between the indexing and retrieval strategies, then the strengths of that store are again diminished.

One challenge across all three research groups is that nearly all tests use a single node configuration. A critical aspect of NoSQL’s scalability comes from being horizontally partitioned across multiple node clusters. By splitting the traffic across multiple nodes subsystems, of the technology can operate in isolation without fighting for finite resources. For instance, an HBase topology brings together process monitoring (Zookeeper), file system management (Hadoop), task orchestration (Yarn), among other background systems. Production environments always distribute these administrative tasks across multiple nodes, specifically because they step on each other during high-stress scenarios. Another concern with single-node configurations comes from the feasibility of continuing to vertically scale up to the demands of the big data workload. The Hadron collider generates 300MB/s (Basanta-Val et al., 2017), making a thirty-day retention 0.78Pb unlikely to fit, let alone be performant on an individual server.

Another limitation that Anikin et al. call out is that their tests use one dataset, and that makes it difficult to conclude generalized observations. This issue also applies to the work of both Martino et al. and Balis et al. NoSQL stores are expert systems that address specific-domains with a specific-access pattern. That introduces complexities to know upfront, *which* storage system in *what* configuration produces the best results. Organizations need to test their workloads at scale across multiple technologies, and then look at the telemetry to make informed architectural decisions.

## Describe the results and contributions

Martino et al. (2019) state that it is equally critical to measure all aspects of the data lifecycle, such as ingestion time, query processing, and storage requirements. Their first test of time to ingest data demonstrated that Mongo and Casandra were multiple orders of magnitude slower than Influx. The second and third shows that Influx is substantially faster to query time sequences than the general-purpose stores, though looking on a non-indexed value was significantly slower than both Mongo and Casandra. These results align with the expectations that purpose-built NoSQL stores constrain the developer’s approach to interacting with the system.

Balis et al. (2017) reduce the risk that an individual query does not align with the data store, by measuring nine variations across millions to billions of records. Their results show that Redis is orders of magnitude faster with query nine taking 0.1 seconds versus Postgres requires multiple 3.2 seconds on the same data size. Though Redis’s in-memory only model limits the tests to ten million records, while Mongo, Postgres, and Influx scaled up to one billion records. The authors also consider the memory usage for each system and call out that Redis followed by Postgres used the most. This result is slightly misleading as Redis keeps the entire database in-memory and Postgres caches frequently accessed pages to improve performance (Madusudanan, 2016). They conclude with a stress scenario of writing 300k records every 30 seconds and executing multiple queries in a loop. These results highlight that Postgre’s table locks are several orders more expensive than the eventual consistency solutions.

Anikin et al. (2019) measure the query time of various graph databases using different topology configurations, such as HBase on a single node versus clustered. Each environment ingested the same data sets, and then the same graph operations (e.g., breath-first and depth-first searches) were performed. The results suggest that a correlation exists between the query duration and traffic patterns. For instance, Postgres was the most aggressive to cache records in memory, and this causes the stress test to perform better after the system warmed up. The researchers also call out the measurable difference between solid-state drives versus hard-disk drives (SSD vs. HHD) for specific systems, due to the slower I/O channel.

# Reproducing the Results

## Describe the data sets used

None of the authors provide links to download the example datasets, nor automation to create identical environments. The Internet has many open-source datasets, and the four specified in Table 1 were selected, as they touch of distinct aspects of the NoSQL data management lifecycle. For instance, the Marvel Hero’s network contains millions of edges, yet is small enough to run locally. In contrast, the L2 Option Pricing is large enough to apply reasonable stress to cloud systems.

Table Data Sets

|  |  |  |
| --- | --- | --- |
| Name | Description | Size |
| L2 US Equity Pricing (HistoricalOptionData, 2019) | Time-series end of day equity quotes – open, high, low, close (OHLC) from 2002 to 2020 | 1 GB |
| L2 US Options Pricing (HistoricalOptionData, 2019) | Time-series end of day quotes plus statistical metadata from 2002 to 2020 | 270 GB |
| Marvel Hero Network (Syntagmatic, 2018) | A property graph of Marvel characters with edges to denote comic appearances together | 17 MB |
| Yelp Data Challenge 2019 (Yelp, 2019) | Subgraph of Yelp reviews, photos, and user information | 8 GB |

## Describe the data stores used

An assortment of different NoSQL stores (see Table 2) was selected to consider the different optimizations across the workloads. When a Docker image was available, those tests were locally performed on a 16-virtual core server with 128GB of memory and 2TB PCIe Gen 3 SSD drive, using the latest stable build as of January 2020. All experiments with cloud-native technologies were limited to the free tier (AWS) and student education account (Azure) restrictions.

Table : Data Store Technologies

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Query Language | Location | Configuration |
| Apache TinkerPop | An open-source graph database | Gremlin | Local Docker | tinkerpop/gremlin-server |
| Azure Data Lake (ADL) | Big Data Batch Processor | U-SQL and C# | Azure Public Cloud | Azure Student Account |
| Influx DB | A time-series database | InfluxQL | Local Docker | influxdb 1.7.2 |
| Athena (+S3) | Serverless Interactive Big Data Query | PartiQL | Amazon Web Services | Default state with a single dedicated S3 bucket |
| Elastic Search | A search store | Lucene | Local Docker | amazon/opendistro-for-elasticsearch |
| Amazon DynamoDB | A distributed key-value | API | Amazon Web Services | Default local table with encryption turned on |
| Postgres | A traditional relational store | PSQL | Local Docker | postgres/postgres:11 |

## What are the observations from data loading

The L2 Option Historical data set comes as one Comma Separated Value (CSV) file per day from February 1st, 2002, through December 31st, 2019 (4509 files). Both Athena and ADL had challenges dealing with the high number of files and required excessive time to start queries. After writing a shell script to concatenate them into a single file, the query start-time improved (hours to minutes). According to the ADL documentation, their system needs to create one processing container per file, and the repeated initialization of that object introduces the noticed lag. The Athena documentation suggests converting records into either Apache Parquet (column-centric) or Optimized Row Columnar (ORC) (row-centric) format before exploration. Using Parquet.NET and approximately one hundred lines of custom C# code encoded the data set and reduce the physical size to 58.5GB from the original 270GB size. Afterward, Athena could handle most filter and group by operations within tens of seconds. ADL had a similar experience where converting the raw CSV files into their internal structured stream format took 50 minutes with 32 tokens, where one token is approximately 2-cores and 4GB of memory. Later queries against the structured stream would complete in 5-10 minutes with four tokens.

The L2 Equity data set was used for local Influx and Postgres containers, both became fully hydrated within ten minutes. Influx Line Protocol is a simple encoding where each line contains a single data point. A local process paginated through the CSV file in blocks of one thousand and then used InfluxDB.LineProtocol 1.1.0 to transmit them. Artificial delays had to be introduced to the local process because Docker would become unresponsive at high loads. A separate process using Npgsql 4.1.2 paginated one thousand lines at a time through the bulk load COPY command and did encounter any issues.

Apache TinkerPop consumed the Marvel Heros data, first using the APIs, and that process took nearly an hour to complete. After writing a script to translate the file into Graph Markup Language (GML), an Extensible Markup Language (XML) dialect, the importation of that file completes in under five seconds. Ingestion of a subset of the Yelp dataset into Elastic Search, using NEST 7.5.1 completed in tens of minutes and appeared to I/O storage-bound. In a production environment scaling the number of writers could have speed up this process. The loading process for Amazon DynamoDB used multiple Amazon Lambda function instances with AWSSDK.DynamoDBv2 3.3.104.23 at near real-time speeds.

## How did the queries differ

Two important aspects of query performance are how long does it take to author queries, and then how long does a typical query run afterward. The times to complete these tasks range widely across technologies (see Table 3). Azure Data Lake was orders of magnitude slower than the other technologies, even when using the local instance and a small data example data set. A lot of time became wasted trying to interrupt poor error messages, such as “Unable to compile script -1” or “Parser ArgumentOutOfRangeException.” After resolving these issues, ADL performed a complex summary that consumed 9.2 TiB of I/O within 7.5 minutes. Running these U-SQL scripts became prohibitively expensive, with two days of experimenting costing nearly 50$. Amazon DynamoDB returns data in near real-time but does not expose any aggregation or data manipulation operators, which results in custom code at every step. At the other extreme Elastic Search with Kibana and Influx with Grafana were both real-time with tooling support to render results in various graphs and tables. The Gremlin interface to Apache TinkerPop was real-time, though its unique syntax is cumbersome and requires a lot of reading even to perform trivial tasks. Athena and Postgres implement the SQL standard, so common operations such as combine (join), aggregate, and filter have a familiar feel.

Table Query Latencies

|  |  |  |
| --- | --- | --- |
| Technology | Time to Author | Query Latency |
| Apache Tinker Pop | 10s of Minutes | 0 to 5 seconds |
| Azure Data Lake | Hours | 10s of minutes |
| Influx | Real-time | 0 to 5 seconds |
| Athena | Few Minutes | 10s of seconds |
| Elastic Search | Real-time | 0 to 5 seconds |
| Amazon DynamoDB | Hours | 1 to 5 seconds |
| Postgres | Minutes | 5 to 30 seconds |

## How flexible are the query engines

Another important aspect of data management systems is their ability to introduce custom functionality and domain-specific operators. Azure Data Lake was the most extensible platform investigated, with support to interweave custom Dotnet code everywhere. For instance, the Yelp dataset used a non-standard JavaScript Object Notation (JSON) encoding, and within fifty lines of code, a custom file parser created. There was also the flexibility to include Dotnet expressions everywhere in U-SQL statements. Postgres exposes limited support to run Python and Perl scripts inside of stored procedures, along with the ability to load third-party extension modules. Athena has limited support to marshal row sets into Java-based Amazon Lambda functions, for scenarios such as pretty printing strings or forwarding into Amazon Sagemaker endpoints. Amazon DynamoDB has a unique strategy, where subscribers can listen to the changelog stream and execute Amazon Lambda functions in response to specific events. The remainder offers some amount of customization, though these are more niche than general-purpose scenarios.

Table : Query Extensibility

|  |  |  |
| --- | --- | --- |
| Technology | Source Code Model | Extension Model |
| Apache Tinker Pop | Open Source | * Java-Based SDK |
| Azure Data Lake | Closed Source | * C# Expressions anywhere |
| Influx | Open Source | * Go API Bindings |
| Athena | Closed Source | * Java-Based Amazon Lambda * Limited AWS Service Integration |
| Elastic Search | Open Source | * Java Plugin Architecture |
| Amazon DynamoDB | Closed Source | * Dynamo Change Streams |
| Postgres | Open Source | * C++ extension modules * Foreign Data Wrappers * Stored Procedures can call Python/Perl |

# What are the conclusions

# Appendix

|  |  |
| --- | --- |
| Technology | Example Query |
| Apache TinkerPop | Find ten characters associated with Wolverine.  g.V(‘Wolverine’) .outE().inV() .distinct().limit(10) |
| Azure Data Lake | Join together two data sets  @daily\_prices =  SELECT UnderlyingSymbol,  MIN(DataDate) AS MinDate,  MAX(DataDate) AS MaxDate,  COUNT(1) AS StockDataPoints  FROM stockholm.dbo.daily\_prices  GROUP BY UnderlyingSymbol;  @options =  SELECT UnderlyingSymbol,  Type,  MIN(DataDate) AS OptMinDate,  MAX(DataDate) AS OptMaxDate,  COUNT(DISTINCT Expiration) AS Contracts,  COUNT(1) AS OptDataPoints  FROM stockholm.dbo.all\_option\_prices  GROUP BY UnderlyingSymbol, Type;  @join =  SELECT dp.\*,  op.Type,  op.OptMinDate,  op.OptMaxDate,  op.Contracts,  op.OptDataPoints  FROM @daily\_prices AS dp  LEFT OUTER JOIN  @options AS op  ON dp.UnderlyingSymbol == op.UnderlyingSymbol;  OUTPUT @join  TO "/metrics/counts.csv"  USING Outputters.Csv(); |
| InfluxDB | Calculate a simple moving average  SELECT MOVING\_AVERAGE(close, 14) FROM quotes WHERE symbol=’MSFT’ |
| Athena | Export all Microsoft Quotes  SELECT DataDate, Open, High, Low, Close FROM stockholm.quotes WHERE symbol=’MSFT’ ORDER BY DataDate DESC |
| Elastic Search | Find Reviews by user 1234 that “contains this” excluding “ignore that” in the same post.   +user\_id:hG7b0MtEbXx5QzbzE6C\_VA file:reviews.json +text:”contains this” -text:”ignore this” |
| Amazon DynamoDB | Get all Microsoft Quotes for last two years  var prices = ddbcontext.QueryAsync<EquityPrice>(  hashKeyValue: "MSFT",  op: QueryOperator.Between,  values: new object[] {   DateTime.Parse("2010-01-01"),   DateTime.Parse(“2020-01-01”)   }); |
| Postgres | Get all Microsoft Quotes for the last two years  SELECT date\_trunc(‘day’,data) as date, (open+high+low+close)/4 as price FROM quotes WHERE symbol=’MSFT’ ORDER BY 1 DESC |