The ScalaTion Time Series Database: Support for Big Data Analytics

by

SANTOSH UTTAM BOBADE

(Under the Direction of John A. Miller)

ABSTRACT

The need to support large-scale time series data is increasing rapidly. There are emerging Time Series Databases built with conventional relational databases or newer NoSQL databases. The ScalaTion Time Series Database is built on top of its column-oriented in-memory database. ScalaTion is an open-source Scala based big data framework for simulation, optimization and analytics. This database provides support for large-scale storage, efficient query processing, pattern matching and a variety of forecasting techniques. Its design goals include the ability to scale up and scale out, and the ability to handle conventional multivariate time series. The database provides an easy way to transform a table into a matrix (or vector) which may be used as input for other data science/machine-learning models that are available in ScalaTion. The capabilities are illustrated via a case study of vehicle traffic forecasting. Multiple experiments are conducted to evaluate the performances of four databases: ScalaTion, MySQL, SQLite, and SparkSQL.

Index words: Time Series Database; Time Series Analysis; Big Data Analytics; NoSQL Database;

The ScalaTion Time Series Database: Support for Big Data Analytics

by

SANTOSH UTTAM BOBADE

B.E., UNIVERSITY OF MUMBAI, INDIA, 2011

A Thesis Submitted to the Graduate Faculty of The
University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2018

The ScalaTion Time Series Database: Support for Big Data Analytics

SANTOSH UTTAM BOBADE

Major Professor:	John A. Miller
Committee:	Krzysztof J. Kochut
	Maria Hybinette

Electronic Version Approved:

Suzanne Barbour

Dean of the Graduate School

The University of Georgia

August 2018

DEDICATION

For Mom and Dad

ACKNOWLEDGMENTS

Firstly, I would like to extend heartfelt gratitude towards my advisor, Dr. John A. Miller. I would like to thank Dr. Maria Hybinette for being part of my committee. I have enjoyed taking Advanced Simulation class under her. She has always being encouraging me in my academics. I would also like to thank Dr. Krzysztof J. Kochut for agreeing to be part of my committe. His constructive feedback, guidance and insights has helped me. A special thanks to all the members of Dr Miller's lab for their interesting discussions throughout my research work. I would also like to thank my friend Prasad Mate for contributing to different aspects of my thesis by interesting and inspirational discussions. Thanks to my family and friends for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As pointed out in [1], research in temporal databases are not focused on time series per se. Although useful, the support and efficiency of using Temporal Databases (TDB) [2] for handling and analyzing large scale time series is lacking [3]. Consequently, Time Series Databases (TSDB) emerged as a new related type of database, starting in the 1990's. The data model for such systems are centered around the concept of *multivariate time series*. A TSDB consists of both ordered and unordered objects (records). Typically, the objects are ordered by time and timestampted (e.g., indicating when the data were collected). There are two main ways that data are collected related to time intervals: discrete-time or discrete-event. In discrete-time, data are collected periodically (e.g., every five minutes for traffic counts from sensors). In discrete-event, data are collected upon the occurrence of events (e.g., traffic accidents that are typically aperiodic).

Research and development of Time Series Databases (TSDB) began in earnest in the early 2000's. This interest lead to the developemt of tools such as STATStream [4] and iSAX [5] for handling time series in the past decades. These tools had a great impact on research, but they were just prototypes to vindicate the research ideas and are no longer maintained. Later, as part of big data landscape, efforts to handle larger and multimodal time series lead to advances in compression as well as parallel and distributed processing techniques. Although static data contribute to the ever exploding amount of data, dynamic data (including time series) are becoming the real issue for big data. Systems like Boeing 787 Aircraft, Internet of Things sensors, etc., generate terabytes (TB) of data every minute [6]. A scalable and efficient TSDB system is needed to cater to the needs of storage and retrieval of such fine-grained data. Such time series may contain billions of data points. Application programmers should be able to find trends and patterns in the historic data and store down-sampled data

points in order to significantly lower the requirements of storage. Traditional databases are not suitable for this type of task.

To illustrate the capabilities of the ScalaTion TSDB, one case study is presented. The case study considers short and medium term forecasting of vehicle traffic on roadway systems. The static data include the roadway system itself, while the dynamic data include traffic counts, statistics on vehicle speed, weather conditions, accident events, sporting events, concert events, road repair events, etc. [7, 8]

The main contributions of this paper are as follows: 1) We present the ScalaTion TSDB, an effective and efficient open-source TSDB that is competitive with existing state-of-the-art TSDBs; for certain operations such as aggregation, projection, selection, intersection and union, the ScalaTion TSDB can be significantly faster than many other open-source databases. 2) The ScalaTion TSDB provides easy integration with a great variety of existing forecasting models provided by the ScalaTion analytics framework without the need to rely on external libraries or systems. 3) Three easy-to-use Application Programming Interfaces, namely Algebraic Interface, SQL-like Interface, and Functional Interface are provided in ScalaTion TSDB to accommodate a diverse group of users who may freely choose their preferred API. 4) ScalaTion TSDB is easy to install and deploy so that researchers may start analytics quickly.

The rest of the paper is organized as follows: Section II discusses the evolution of Time Series Databases, starting with their predecessor, Temporal Databases, as well as Spatial-temporal Databases. The Related Work in Section III highlights the state-of-the-art Time Series Databases and discusses the support of present day systems for time series analysis. The architecture and implementation of the ScalaTion Time Series Database with emphasis on scaling up and scaling out are discussed in Section IV. Three Programming Interfaces are presented in Section V. The performance of the ScalaTion Time Series Database on real

datasets is evaluated and compared with modern day alternatives in Section VI. S Finally, contributions of this work, conclusions and future work are given in Section VII.

CHAPTER 2

BACKGROUND

In the late 1980's, a new type of database systems emerged to support temporal attributes and enabled users to store temporal data. About two decades worth of research has been dedicated to make temporal data modeling, storage and querying expressive and efficient [9]. The efforts in academia [5] and the industry [10, 11, 12] lead to the systems which are popularly known as Temporal Databases.

2.1 Temporal Databases

Temporal Database (TDB) is a database system which records with associated time ranges, i.e., start timestamp and end timestamp, which may denote a valid time (time for which a fact was valid in the real world), a transaction time (time duration for which a fact existed in the database) or bi-temporal (supports both valid and transaction time). TDBs are designed to manage historic data, i.e., records with the different versions as opposed to a database which manages only the current state of its records and to provide a temporal query language to query such data. However, with the rise of modern Internet of Things sensors and long sequence of data points generated by them at regular (and typically short) intervals [6], TDBs need to scale in order to store and process such data. Due to the limitations of underlying relational model, TDBs generally do not scale well [13]. To address the issues of scalability, NoSQL Temporal Databases were proposed as an alternative [14, 15].

Time-series data is a special case of temporal data but has its own specific characteristics. There usually is no specific pattern in the temporal data stored in the traditional temporal databases where as time series data have seasonal trends, patterns, and are expected to be timestamped at a constant time gaps. Thereby they open up scope for analytics. Time series data have specific modeling and functional requirements which are not met by the design

of traditional TDBs. Even with the NoSQL design, not all TDBs provide efficient ways of transforming time oriented data by applying mathematical and statistical techniques such as moving average and other complex transformations [16]. These design issues, scalability issues, time series analysis requirements motivated the development ScalaTion TSDB.

2.2 Spatial-Temporal Databases

A Spatial-Temporal Database (STDB) can be seen as an extension to the temporal database with added support for spatial attributes. GPS navigation systems, health monitoring systems, GIS, etc., need to store and process sequences of billions of data points over time along with the changes in the objects' locations/geometry. A traditional RDBMS solution is not suitable for such needs because the system should be able to scale and capture more than one dimenion [17].

CHAPTER 3

Related Work

There are many Time Series Databases available today with varying underlying base technologies and level of support for time series analysis. Table 3.1 lists a few of them.

Table 3.1: Modern Time Series Databases

Name	Organization	Base
ScalaTion TSDB	UGA	Columnar DB
OpenTSDB	Yahoo!	HBase
InfluxDB	InfluxData Inc.	TSM Tree
Druid	Apache 2.0 Community	Columnar DB
Gorilla	Facebok	ODS

OpenTSDB is based on HBase. InfluxDB data storage model is a custom LSM tree-based approach called Time Structured Merge (TSM) tree. Gorilla [18] has its own Operational Data Store (ODS), an important part of the monitoring system at Facebook is built using HBase. ScalaTion TSDB and Druid are column-oriented databases. While OpenTSDB and InfluxDB store data on-disk, ScalaTion TSDB, Druid and Gorilla are in-memory systems. Query execution is slower when performing on data that is on disk than when working on data in memory. Thus, in-memory systems take advantage of producing faster results.

The systems are accessed with the APIs and interfaces they provide. OpenTSDB, InfluxDB and Druid provide REST HTTP APIs and GUI/CLI to connect and execute queries. Gorilla has its own SQL-like interface. ScalaTion TSDB has multiple interfaces and provides APIs for the Scala language to execute queries. More on the interfaces is discussed in Section V.

Working with large amounts of data requires large amounts of disk space / memory and querying on these large structures could be time-consuming. Therefore, major TSDBs include compression techniques in storing the data and support performing operations on compressed data. OpenTSDB performs compression using the LempelZivOberhumer (LZO) algorithm

that is included in HBase. Druid uses a bitmap compression algorithm to reduce the size of data. Gorilla achieves compression using delta encoding on XOR comparisons with previous values. In ScalaTion TSDB, we use Run Length Encoding to compress the columns of the relation. This is further discussed in detail in Section IV.

The availability of basic relational algebra operators and aggregate functions, such as, AVG, COUNT, MAX, MIN, SUM, etc., in the system is helpful in performing tasks on the data. This reduces the effort of users to write the functions for commonly used operations. More on relation algebra support in ScalaTion TSDB is discussed in Section V. Table 3.2 shows support of each of the TSDBs for various queries. π - PROJECT, σ - SELECT, $\bowtie$ - JOIN, Φ - aggregate functions.

Table 3.2: Query support in TSDBs

Name	π	σ	$\bowtie$	Φ
ScalaTion TSDB	✓	✓	✓	✓
OpenTSDB	✗	✓	✗	✓
InfluxDB	✓	✓	✗	✓
Druid	✗	✓	✗	✓
Gorilla	✗	✓	✗	✗

The frequency (sampling rate) and phase may be used to define the distance between two timestamps. Data with high sampling rates require huge memory space to store the data. Popular systems such as OpenTSDB, and Druid can not support a sample rate faster than 1 ms. While, ScalaTion TSDB supports sample rate of 1 ns. All these TSDBs also allow users to choose the sampling rate depending on the need so that data storage space is reduced.

Studies have shown that for certain cases, using a TSDB yields better result than using a traditional database [3]. TSDBs also find their use in data mining techniques like search, classify and clustering. Most of these TSDBs support analytics and metrics on time series data. The analytics interface in ScalaTion is discussed in Section V. Detailed comparisons

between major TSDBs and those with selected RDBMSs are discussed in Bader A., et al. [16].

With the increase in applications using time series data in fields like IoT, IT, monitoring systems, and many more, multiple time series analytics libraries are being developed for different statistical platforms. Time Series Analysis of R [19], StatsModels for Python [20], Spark-TS and Flint for Spark to name few such libraries. Also to notice, most these libraries are third-party and not in-built in the toolkit. While, The ScalaTion TSDB leverages upon its underlying libraries, analytical packages and forecasting models provided natively within the framework.

Platforms such as Spark, R, Python have support for dataframes. Dataframes are table like structures to store data with equal sized columns and columns can be of different data types. Like most other platforms, ScalaTion provides dataframes API called as RelationFrame or RelationF in short [21] and is discussed in Section V. They provide a convenient way of performing big data analysis.

CHAPTER 4

THE SCALATION TSDB

The ScalaTion big data framework currently consists of four major modules: `scalation_mathstat`, `scalation_database`, `scalation_modeling` and `scalation_models`. A fifth, `scalation_automod` is under development. Within the `scalation_database` module, the ScalaTion TSDB is in the `timeseries_db` package which is a subpackage of the `columnar_db` package. There are many modeling techniques in the `scalation_modeling` module that utilize the `timeseries_db` package. ScalaTion TSDB uses its custom `TimeNum` class wrapped around Java's `Instant` to store date-time information along with a time-zone [22]. The decision to select `Instant` was driven by the need to support a sampling rate as low as nanoseconds, to store data relative to UTC (Universal Time Coordinated) so that it does not have impact of Day Light Savings time. Also, storing all the temporal data in the UTC zone makes the arithmetic operations easier and independent of the user's local zone, from where the framework is being used. However, ScalaTion gives flexibility to its users to change the default time-zone for the current session and then load the temporal data adjusted to that zone. Instant is a light weight choice as it stores only the epoch seconds and part of the current second, and uses only 12 bytes per timestamp. So this choice was obvious compared to heavier APIs such as ZonedDateTime or LocalDateTime which needs 29 and 15 bytes, respectively, per timestamp.

The ScalaTion TSDB rests on top of the underlying in-memory columnar database. Time-series data may be huge in size and loading it on-demand from the disk can be very time consuming. For this, ScalaTion TSDB loads data into relations for the first usage and then keeps all the loaded relations in-memory for the current session. The ScalaTion framework provides a SQL-like shell to load, save to disk and query relations. User can write their custom DDL, DML queries in `Scala` objects and pass them through this command

prompt to execute the queries contained within them. Work is on the way to provide the ScalaTion TSDB as a microservice [23].

4.1 Underlying In-Memory, Columnar Database

Underlying the ScalaTion TSDB is an in-memory, column-oriented database system [21] that has been designed to support advanced analytics. In analytics, the choice of a columnar database is natural as Online analytical processing (OLAP) queries rarely need to access entire rows. Columns in a columnar database database may be efficiently extracted to form vectors or matrices. In particular, when performing time-series analysis, a columnar architecture can provide benefits of processor's L1, L2 and L3 caches as typical forecasting techniques, such as ARIMA (Auto-Regressive Integrated Moving Average) needs to access past data points in a column.

For efficiency, preprocessing of data may be combined with data extraction. Preprocseing techniques include, handling missing values (e.g., through data imputation), converting strings to integers, removing outliers or even rescaling the data. Once the processed data are stored in the data structures from the `linalgebra` package from the `scalation_mathstat` module, many types of analytics/machine learning algorithms may be applied.

4.2 Compression

Time series data are bulky in size. Columnar architecture can effciently compress data in columns. Time-series data can have a great scope for compression since, many of the column values, such as sensor ID, sensor location, etc., do not change over time. These characteristics of the data demand for a compression technique that allows performing analytics on the compressed data, because the cost of decompression may dominate overall analytics performance [24] and decompressed data might be too big to fit in the memory. Time series data are particularly suitable for compression. The ScalaTion TSDB utilizes the compression techniques present in the underlyting columnar database [25]. It is a norm to have times-

tamps at regular intervals, ScalaTion TSDB stores the delta of timestamps to compress continuosly changing time column. Thus, its underlying Run Length Encoding (RLE) compression can significantly reduce the size of the time column. The actual non-temporal values can be compressed using RLE or a compression technique presented in [18].

4.3 Scalability

In order to support big data and high performance analytics, ScalaTion provides scaling up in standalone mode and work on scaling out in cluster mode is ongoing.

4.3.1.0 Scaling Up

As a big data framework, ScalaTion emphasizes ease of installation and use. In standalone mode, ScalaTion may be simply downloaded and run, for example, using the Simple Build Tool (SBT) or in the Eclipse/IntelliJ IDE. Performance is provided by careful coding of algorithms, multithreading and initial support for vector instructions. As an example of careful coding, matrix multiplcation can be coded to maximize the effectiveness of memory caches [26]. Starting with the naïve triple loop implementation, we obtained a speed-up of 4 times by taking transpose of the second matrix. We observed further improvement by 20% when 2D matrices were stored into 1D arrays. In addition, the use of block matrices, increases the effectiveness of multi-threading. Our naïve implementation block matrix multiplication improves multicore performance by 50%, while with our optimized SUMMA block matrix multiplication algorithm [27], we observed, upto 3 times speed over our naïve blocked parallel implementation. These results are further discussed in detail in the Section VII. We have also shown how we can optimize our algorithms to make efficient utilization of cache memory. Our algorithm based on dgemm of BLAS has been implemented and performance of different block sizes has been evaluated. We observed that, block size should be large to have better performance as it will ensure more cache hits. However, increasing block size beyond the cache size causes cache misses and thereby does very little to improve performance. Finally, the use of vector instructions such as Intel's AVX-512 instructions is currently provided via

the `blis` library as an option. The `blis` libary is one of the more portable Basic Linear Algebra System (BLAS) libraries available today [28]. A design goal is to make ScalaTion a pure JVM solution and not use native libraries such as `blis`. Unfortunately, the current JVM does not support Advanced Vector Extensions (AVX) [29], although there appears to be some work on adding them [30]. In standalone mode, big data is supported in two ways: The use of efficient data structures in servers/workstations with large main memories, or out-of-core capabilities through the use of memory-mapped files.

4.3.2.0 Scaling Out

For datasets too large to fit in a single main memory or for which the memory mapped file are too slow, ScalaTion may be deployed in a cluster. The intent is to keep installation, deployment and execution relatively simple. Although ScalaTion may use a distributed file system such as the Hadoop Distributed File System (HDFS), it is designed to rely on maintaining big data in memory à la Spark [31].

Similar to other big data frameworks, ScalaTion uses message passing for communication between compute nodes in the cluster. Although Distributed Shared Memory provides higher abstraction and a more convenient programming environment, performance issues have kept it from widespead apdoption [32]. Providing reliable and efficient message passing in a cluster requires sophisticated software. Spark previously used Akka [33], but now uses 'org.apache.spark.rpc' [34] which uses Netty and Java NIO and provides a Akka like interface, but making application development more flexible to the users.

The ScalaTion framework can be made to use the default Akka ActorSystem in order to provide the dispatcher for each Future's ExecutionContext. However, any Akka actor system can be used, including a distributed one via via Akka Cluster [35].

4.4 Support for Time Series Analysis

Time series data is, usually, collected through small sensor devices. And such data is prone to have consistency and accuracy issues, such as, irregularities in time-stamps because of network latency or bad sensors, certain data points might get lost over the network channel or might get corrupted to value far-off from the actual recorded value or might incorrectly send the same values for a long continuous time, etc. Junk values produces junk analytical results. The ScalaTion TSDB provides variety of preprocessing techniques to handle such data. Time-series data is expected to be received at fixed intervals, however, as mentioned above, we may have a few or more data points received at an expected time stamp. A time-series database should be able to align such time stamps with the nearest expected time stamp. Also, it is highly probable that 2 different time-series data collected from 2 different types of sensors (say, `weather` and `traffic` will not be synchronized on time stamps, either due to they have a different sampling rate or if they have same sampling rate, they might be clocked at different instant of time (say, different minutes of hour). The ScalaTion TSDB provides a convenient API, called as `leftJoinApx` to handle such cases. It needs to be called as,

```
r.leftJoinApx (tPosL, tPosR, s)
```

where, `r` and `s` are `Relation` objects, which stores time stamps at index `tPosL` and `tPosR` in them, respectively. This function treats two timestamps as equal if they are at certain threshold distance from each other. ScalaTion supports threshold of as minimum as 1 nanosecond and this threshold can be set before invoking `leftJoinApx` as,

```
TimeO.setThreshold(secs.nanos)
```

Outliers are data values which lie at an unexpectedly greater distance from other values from the dataset. Analysis performed on data which has outliers may be misleading. As mentioned above, time series data may have outliers because of bad sensors or values getting

corrupted over while in transmission. ScalaTion provides a variety of techniques to remove outliers and mark them as 'Missing'. Later, such marked missing values or the missing values already present in the dataset can be imputed using `Imputation` methods provided within the framework. ScalaTion provides flexible APIs for both, removing outiers and imputing missing values, where users may select framework's out-of-the-box techniques, such as `Linear Interpolation`, `Moving Average`, for `Imputation` and `Standard Deviation`, `Percentiles` for `Outliers` etc., or provide their own method to detect outliers and impute missing values.

```
Outliers.rmOutliers
(relationObj, column,  method)
Imputation.replaceMissingValues
(relationObj, column, MisingValue, method)
```

Time series data may contain several million to billion data points. It may not be feasible to store all the data points. Especially, data points from beyond certain past might not be useful for analysis. Downsampling refers to lowering down the sampling rate of a time series. ScalaTion provides an easy way to downsample time series data through it `downsample` API. It can be used as,

```
relationObj.downsample (seconds.nanos)
```

Our TSDB supports, downsampling to as low as a few nanoseconds, i.e., say, if an application records data points at every 2 nanoseconds then it can be downsampled to a time series with a sampling rate of 3 or more nanoseconds with this API.

Some domains data analysis may not be able to produce more insights or might not be interested in certain minutes, hours, days, etc., For example, the traffic forecasting case study discussed in this paper does not consider weekend data as weekend data has different pattern that that of weekdays data and also, data from 7:00 PM to 7:00 AM is of relatively less important since it has less congestion compared to that of rest of the hours in the day.

ScalaTion TSDB provides `removeChrono` API, which can be used as follows to remove all timestamps which has unimportant seconds, minutes, hours, days, etc.,

```
relationObj.removeTime (ChronoField.FIELD,
    Seq (Int Values))
```

where, `ChronoField.FIELD` is a constant from the set of fields in `Java`'s `ChronoField` enumeration and `Seq (Int Values)` are the `Int` values of the `FIELD` under consideration.

It is not uncommon in time series data to come across sensors which are not worth for analysis because they of the abnormally great number of missing data present sent by them. Such sensors contribute very little to the analysis and thus, can be discarded. ScalaTion TSDB provides a flexible API to determine if certain time series is good enough or not. It can be used as,

```
relationObj.maxThres
  ( ColPos, MissingVal, Thresh)
```

CHAPTER 5

APPLICATION PROGRAMMING INTERFACES

The ScalaTion TSDB provides Three Application Programming Interfaces (APIs).

5.1 Algebraic Interface

The first API is an extended relational algebra that includes the standard operators of relational algebra plus those common to column-oriented databases. It consists of the `Table` trait and two implementing classes: `Relation` and `MM_Relation`. Table 5.1 shows the thirteen operators supported (the first six are considered fundamental). Operator names as well as Unicode symbols may be used interchangeably (e.g., r *union* s or $r \cup s$ compute the union of relations r and s. Note, the extended projection operator *eproject* (Π) provides a convenient mechanism for applying aggregate functions. It is often called after the *groupby* operator, in which case multiple rows will be returned. Multiple columns may be specified in *eproject* as well. There are also several varieties of *join* operators. As an alternative to using the Unicode symbol when they are Greek letters, the letter may be written out in English (*pi*, *sigma*, *rho*, *gamma*, *epi*, *omega*, *zeta*, *unzeta*).

5.2 SQL-like Interface

Although the algebraic interface is fully capable, users familiar with SQL, may prefer the ScalaTion SQL-like interface provided in the `RelationSQL` class. The following SQL query retrieves the average traffic count for sensors located on road segment "rs1".

```
select sensorId, sensorName, avg(tcount)
from roadSegment natural join trafficData
where rsName = 'rs1'
group by sensorId
```

Due to the flexible syntax on Scala, this can be nearly reproduced directed in the language.

Table 5.1: Extended Relational Algebra (r = roadSegment, s = sensor, t = trafficData, q = tollRS)

Operator	Unicode	Example	Return
select	σ	$r.\sigma$ ("rsName" == "rs1")	rows of r where rsName == "rs1"
project	π	$r.\pi$ ("rsName", "sensorId")	the rsName and sensorId columns of r
union	$\cup$	$r \cup q$	rows that are in r or q
minus	-	$r - q$	rows that are in r but not q
product	$\times$	$r \times t$	concatenation of each row of r with those of t
rename	ρ	$r.\rho$("$r2$")	a copy of r with new name $r2$
join	$\bowtie$	$r \bowtie t$	rows in natural join of r and t
intersect	$\cap$	$r \cap q$	rows that are in r and q
groupby	γ	$t.\gamma$ ("sensorId")	rows of t grouped by sensorId
eproject	Π	$t.\Pi$ (avg, "tcount")	the average of the tcount column of t
orderBy	ω	$t.\omega$ ("sensorId")	rows of t ordered by sensorId
compress	ζ	$t.\zeta$ ("tcount")	compress the tcount column of t
uncompress	Z	$t.Z$ ("tcount")	uncompress the tcount column of t

(roadSegment *join* trafficData)

.*where* [String] (("rsName", _ == "rs1"))

.*select* ("sensorId", "sensorName")

5.3 Functional Interface

With the increasing popularity of functional programming along with its use in big data analytics [36] and its excellent support by the Scala language [37], it is straightforward and useful to provide a functional API to access ScalaTion TSDB databases. The `RelationF` class provides the standard *map*, *reduce*, *fold*, *filter*, *groupBy*, *orderBy* and *join* high order functions. The same query can be written in functional style as follows:

(roadSegment *join* trafficData)

.*filter* [String]("rsName", _ == "rs1")

.*map* [Int](_ / 10, "tCount")

.reduce [Int](_ + _, "tCount").show()

CHAPTER 6

PERFORMANCE EVALUATIONS

6.1 Vehicle Traffic Forecasting

Forecasting is the art of taking available information of the past and attempting to make the best educated guesses of the ever unforeseen future. From the historical data, patterns can be observed and forecasting models have been developed to capture such patterns. This case study focuses on forecasting traffic flow in major urban areas and freeways in the city of Austin, TX using large amounts of data collected from traffic sensors. As an extended study, we will also be incorporating precipitation data into forecasting models to better predict traffic flow in rainy weather.

6.2 Test setup

We used UGA's Sapelo cluster for evaluating the performance of the proposed and discussed systems. Sapelo is the computing environment provided by the Georgia Advanced Computing Resource Center (GACRC) [1]. The tests were run on compute nodes having 48-core AMD Opteron processors and 128GB of memory to facilitate parallel processing and handling large datasets in-memory. For evaluating performance of the database systems discussed, we used the traffic data provided by City of Austin Transportation Department [2]. This dataset has 15 minutes resolution data collected from several sensors across the city of Austin, from the June 2017 to the current date. Even higher resolution data, recorded every minute or 30-seconds, is able to capture the dynamic nature of traffic patterns in major urban areas and freeways. The Caltrans Performance Measurement System (PeMS) from the state of California is one such dataset but PeMS does not let users download data without creating an account. We decided to not use datasets provided by PeMS following

[1] https://gacrc.uga.edu/

[2] https://data.austintexas.gov/Transportation-and-Mobility/Travel-Sensors/6yd9-yz29

the open science philosophy, mentioned in [38]. We used the precipitation data provided by the Automated Surface Observing System (ASOS) [3], a joint program maintained by the National Weather Service (NWS) and the Federal Aviation Administration (FAA). The ASOS sensors are typically placed in airports or air bases. Data are downloaded in hourly resolution through a convenient web interface [4] provided by the Department of Agronomy of Iowa State University. Out of the 3 available weather sensors, EDC (Austin Executive Airport), AUS (AustinBergstrom International Airport) and ATT (ATT Austin Airport), we selected only the AUS and ATT sensors for the evaluations on traffic forecasting case study. We decided to not include EDC sensor as it was located about 90 miles from our traffic sensor. Also, to test our scale-up and scale-out capabilities we have used generated synthetic data. We have also evaluated database query operators provided by ScalaTion with the other three popular open source databases, namely, MySQL Server (Version 5.7.20), SQLite (Version 3.25.0), and, Apache Spark (Version 2.3.1).

6.3 Scale up performance of Linear Algebraic Operations

Matrix multiplication is one of the most frequently used operation in various data analytics and machine learning models. Years of research has been devoted in optimizing it. With the advent of multi-core parallel machines and distributed networks to handle larger datasets, effective algorithms for multiplying matrices and other core linear algebra operations such as dot product of vectors, with such scale up capabilities are needed. We have presented one algorithm for multiplying matrices which enhances cache optimization by effectively utilizing L3 cache memory and one algorithm for blocked matrix multiplication. We have tested this on square matrices with perfect square size dimensions so that our naïve blocked implementation can be easily divided into square blocks of size $\sqrt{dimension}$. As can be seen from Fig 1. and from Fig. 2, SUMMA [27], Scalable Universal Matrix Multiplication Algorithm, out-

[3] https://www.weather.gov/asos/

[4] https://mesonet.agron.iastate.edu/request/download.phtml

performs other other algorithms for larger input sizes. From Fig. 1, we also observed that the implementation which supports commonly used types of Matrices in ScalaTion through the use of generics, is about 10 times slower than the same naïve implementations with specific types. This motivated the need for code generation classes with the support for different base types. The number of cache misses in the traditional naïve triple loop implementation can be reduced by taking the transpose of one matrix and then multiplying rows by rows, instead of rows by columns. As a result, 4-5 times speed up is observed. From Fig. 1 and Fig. 2, it can also be seen that use of blocks, if done correctly, can increase the performance for larger matrices. Fig 2. shows the performance of the implementations dicsussed in Fig. 1.We observed similar relative trends among implementations as that in their serial versions. As we can see, SUMMA is better suited for paraller processing of larger data size. Fig. 3 and Fig. 4, shows the results for the implementations for matrices stored as 1D array. We have shown speed up of using vector instructions set in C throgh `blis` library. A standalone `blis` by itself gives faster implementation on the lines of BLAS, however this performance gain is compensated by the JNI overhead in copying the data to and back from the C code, and loading the native code library into JVM. JNI overhead is linear to the size of the input and thus it discouraged to use native libraries, if fewer and less complex calculations are to be performed on greater data. Since, matrix multiplication is a cubic operation, we can see that `blis` starts with a 4 times speed up for smaller input and increases it to a 8-10 times speed up when computation costs are much higher than the JNI overhead. We, at the time of writing, do not have 1D counter part of SUMMA algorithm,however, efficiency of parallel processing by the use of block matrices for large input sizes can be seen through our naïve implementation of block matrices. Moreover, a C/ C++ version of implementation of this code auto-vectorizes itself, so we are sure that when it is introduced in the JVM, we shall obtain a greater speed up. For readability and better understanding, the actual running

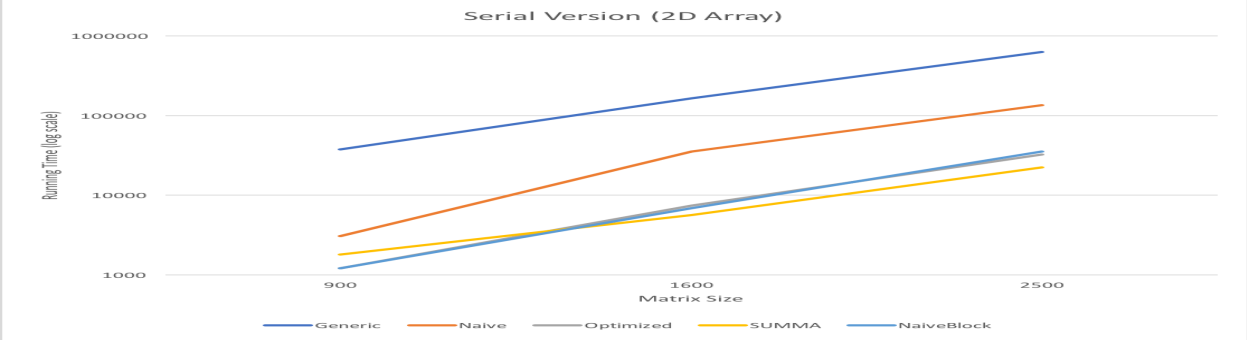

Fig. 6.1: Running times of serial version of square matrix multiplication

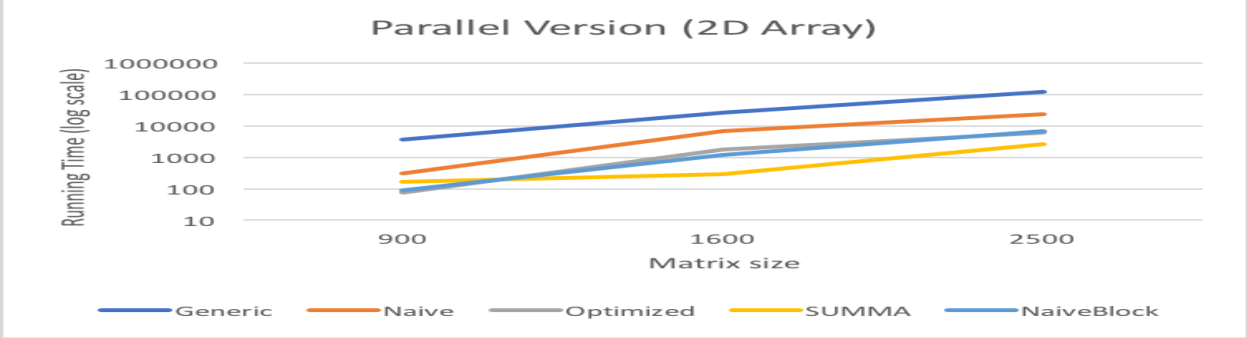

Fig. 6.2: Running times of parallel version of square matrix multiplication

times for scale up performance of matrix multiplication are given the TABLE 7.1, 7.2, and 7.3 in the Appendix.

6.3.1.0 Evaluations of Databases

In this subsection, we evaluate ScalaTion database with other popular open source databases. We have tested 4 of the common operations in the analytics databases, namely, Functions, Union, Indexed Join, and Intersect.

We have evaluated sum, average, min, max, etc., aggregate functions on the databases mentioned in the Fig 6.5, all of the functions produced similar results. So we decided to take

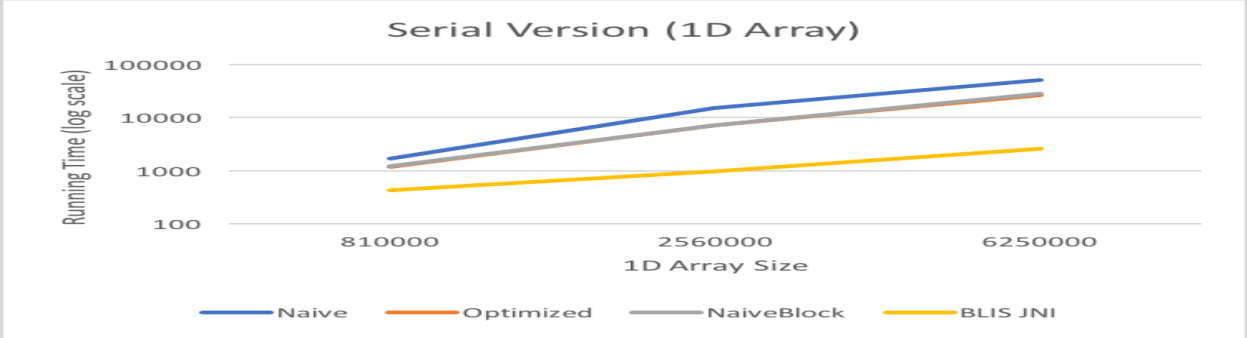

Fig. 6.3: Running times of serial version of square matrix multiplication

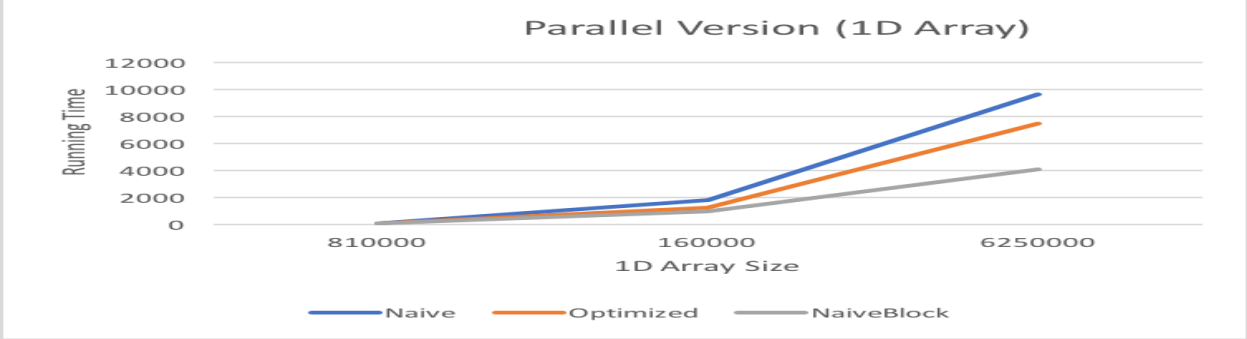

Fig. 6.4: Running times of parallel version of square matrix multiplication (1D Array)

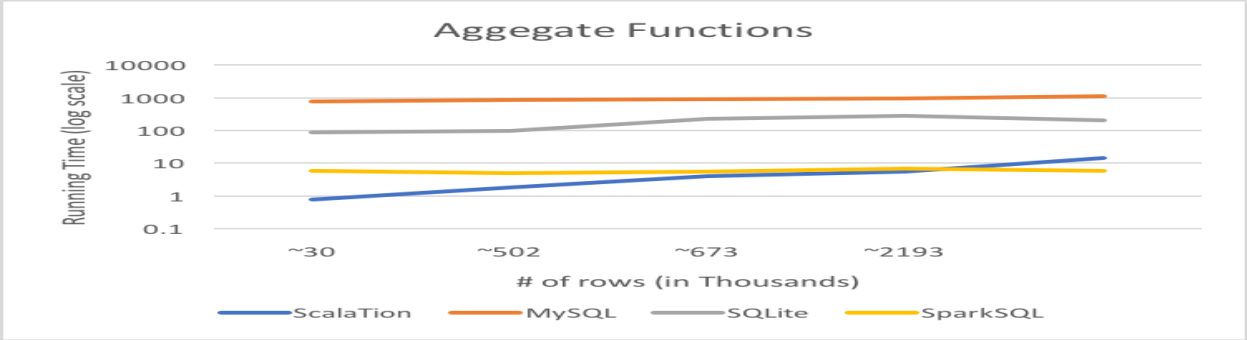

Fig. 6.5: Aggregate Functions Performance

`avg` as a representative function to find average traffic on a lane, a highway or entire city of Austin. ScalaTion outperforms all the other databases compared by on the small to medium sized dataset while its performance is comparable to the `SparkSQL` on the larger sets.

Then we tested these databases for a `Union All` query and the results are plotted in 6.6. Austin traffic data is collected across individual lanes of highway, so a Union query can be used to aggregate data across lanes of such multi-lane street or highway. As can be observed from the 6.6, ScalaTion outperforms other databases for medium sized databases and its performance is comparable to the `SparkSQL`.

We evaluated an indexed join query as join is one of the most important operation in the databases, since it is rare to have all the data within only one table. We decided to join Austin's `traffic` sensor data with the nearest `weather` data, to get insights into correlation betwen traffic volume and measured precipitation at the same time. From the Fig. 6.7, we observe that ScalaTion performs better than `MySQL` and `SQLite` while it does not perform as well as `SparkSQL`. After investigation, we have found that, `SparkSQL` uses a custom implementation of `OpenHashMap` hash-join for their `join` operation and currently,

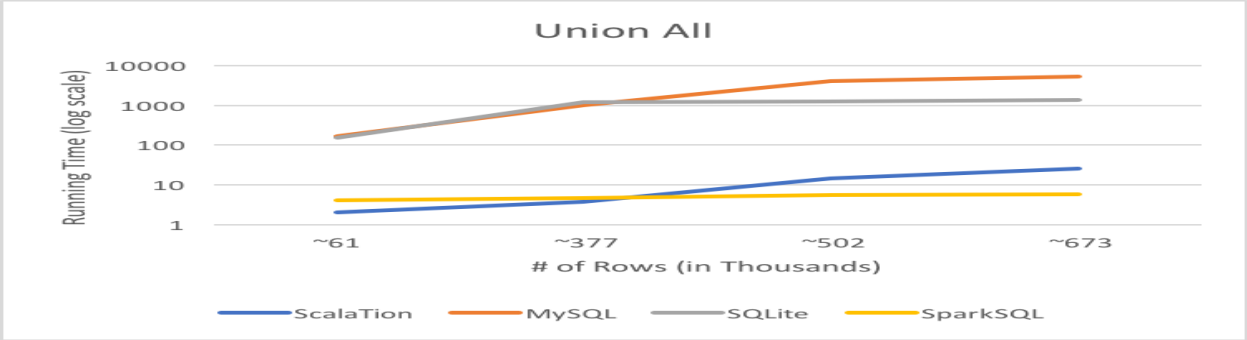

Fig. 6.6: Union Performance

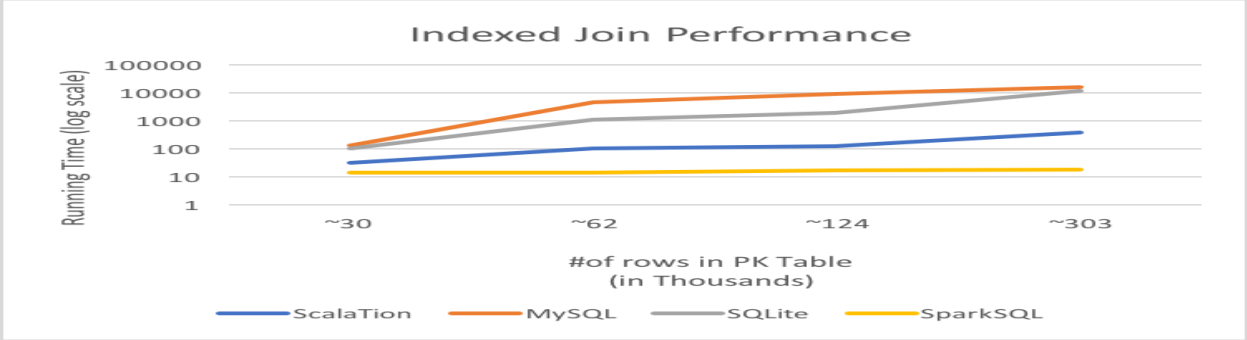

Fig. 6.7: Indexed Join Performance

efforts are being made to provide our own custom implementation of Hash-Map like structure for improving efficiency of hash-join.

Finally, we compared `intersect` operator, as it is quite common practice to query the database, especially in the analytical database, to find common records in multiple relations. For example, we queried the 2 weather sensors located at 15 miles from each other, to find what times they both recorded same rain activity. Results in 6.8 shows that ScalaTion performs better than the row-stores `MySQL` and `SQLite`.

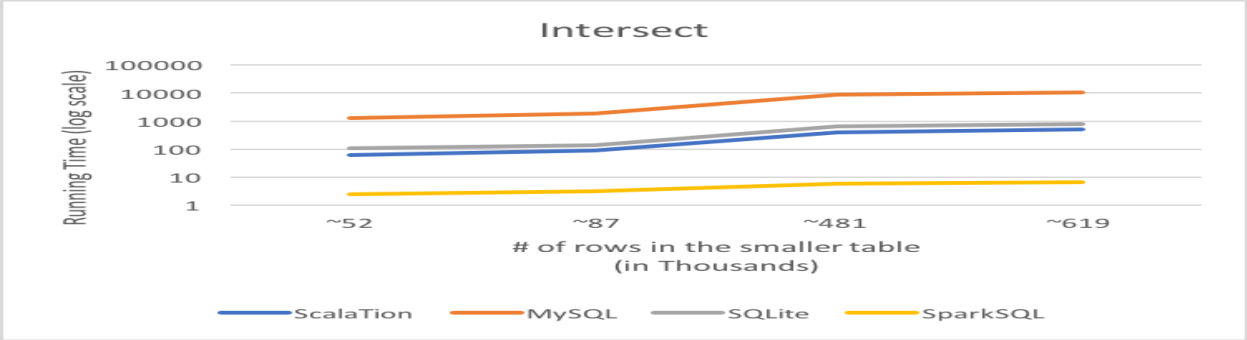

Fig. 6.8: Intersect Performance

It should be noted that, even though `Join` and `Intersect` operations are more suited for traditional row-store RDBMSs, `ScalaTion`, despite being a column-store, convincingly outperforms them. Also, since the data were time-stamped at irregular intervals, the compared systems did not give correct joined or intersected result. `Spark` depends upon external `Spark-TS` or `Flint` library to align such nearby time-stamps [39], [40], where as, ScalaTion has support for this out-of-the-box.

6.4 Evaluations on Compression

As discussed in the SECTION IV above, time-series datasets can be huge in size and columnar-store can efficiently provide compression to save on space required for storage. We used Run Length Encoding (RLE) compression technique provided by ScalaTion to compress the weather dataset generated by 2 sensors, namely `ATT` and `AUS`. As can be seen from the TABLE 6.1, we have accomplished to compress non-temporal attributes such as, station, lat, long tremendously and could compress 1,197,717 values to just 6 values, while other attributes which does not change frequently over time, such as, `visibility`, `precipitation` are also compressed to 10.88% to 14% of the space for uncompressed raw data. ScalaTion uses its custom `TripletD` *`(value, count, startPos)`* data-structure to store run length

encodes. Each `TripletD` element takes 16 bytes of memory. Space saved in terms of bytes is shown in the TABLE 6.1.

Table 6.1: Weather Dataset Compression

Column	Original	After Compression	In %
station	399239	2	0.0005009
valid	399239	394378	98.78
lon	399239	2	0.0005009
lat	399239	2	0.0005009
temperature	399239	149256	12.33
dew point	399239	129683	32.48
relative humidity	399239	161320	40.40
wind direction	399239	225323	56.43
wind speed	399239	247029	61.87
precipitation	399239	57162	14.31
visibility	399239	43468	10.88

As can be seen from the TABLE 6.1, we end up wasting space for the compressed time-stamp column, and it takes almost the double the space as that of the original uncompressed raw time column. Time-stamps are unique and thus, are not a good candiate to compress using `RLE`. We have used only 59.55% of the space for the entire uncompressed weather dataset.

6.5 Evaluation of Traffic Forecasting for Austin City

Traffic data are collected for Austin, Texas [5]. Sensors are placed at 86 locations across the city and data are collected using a single detector per lane. The collected traffic data include volume, occupancy (% capacity of the lanes) and speed. Forecasting volume is of primary interest in this case study. Data are generally available from summer/early fall of 2017 and are being continually collected till the present time. In this case study, the end date is set to the 1st of July, 2018.

The data are only available in a very raw format and much pre-processing is needed before the data can be used for analytics. A single data file contains the data for all the detectors, but the rows in the data file are not in any sorted order. As the first step, the data file is loaded

[5]https://data.austintexas.gov/Transportation-and-Mobility/Travel-Sensors/6yd9-yz29

into a `RelationSQL` object and sorting is performed by Detector ID and then by time stamp. The data are in 15-minute intervals, but unfortunately a data row may not exist for every 15-minute interval. To make this dataset more suitable for analytics, an approximate left join is performed using a `RelationSQL` object containing only a single column of regular 15-minute intervals with another `RelationSQL` object that is produced by a select query based on a particular Detector ID. The approximate join operation is needed because the original data file does not always contain regular, perfect 15-minute interval, time stamps. The resulting `RelationSQL` object is one with the regular 15-minute time interval but contains essentially blank rows which are not available in the original data file and must be imputed in a later step. At this point, certain time series produced by Detector IDs simply contain two many blank rows. Therefore, any detectors that contain more than 20% of blank rows or contain only 20% or less observations than the detector with the most observations are removed from consideration. ScalaTion provides an API to perform any of the above mentioned things. Furthermore, the remaining time series from different detectors are further filtered tout based on the number of zeros, which are used to represent missing values in this data file. Unfortunately, it is impossible to distinguish an actual data value of zero and a missing value represented by zero in this particular data file. To be on the safe side, all values of zeros are imputed; all the blank rows, representing the originally missing time stamps, are imputed as well. As discussed in the previous section 4.4, ScalaTion provides a `MissingVal` paramter in the `replaceMissingValues` API, this parameter can be set to zero or any value which is to be considered as 'Missing' and will need to be imputed. The presence of outliers is also checked. Outliers are removed and imputed if any can be found. In the end, time series from 47 detectors out of 86 can be used to perform analytics.

Among the remaining data, only data from weekdays are considered since data from weekends are of relatively less interests and they typically exhibit different traffic patterns than weekdays. This is a commonly found practice, as can be seen in recent work in [41], [42],

among others. Furthermore, only day time hours from 7:00AM to 7:00PM are considered for forecasting since traffic patterns during this time can exhibit the greatest congestions. As dicsussed in the section 4.4 we used our framework's API `removeChrono` to remove data of weekends as well as weekdays' hours from 7:00 PM to 7:00 AM.

Several commonly used forecasting models are used in this case study, including the ARIMA (Auto-Regressive Integrated Moving Average) [43] family of models, Exponential Smoothing [44, 45] and feedforward, fully connected Neural Networks. In the ARIMA family, three models were used, an ARIMA model whose order is chosen by an automated search using the AICc (Akaike Information Criterion) criterion as described in [46], a Seasonal ARIMA (SARIMA) model using a similar automated search for orders and the SARIMA$(1, 0, 1) \times (0, 1, 1)$ model that has been successfully applied in traffic forecasting in several related work [47, 41, 8]. The Exponential Smoothing uses additive seasonality, as suggested in Section 7.5 of [48], since no clear multiplicative seasonal pattern is present for traffic data. The Neural Network structure consists of an input layer of 96 neurons, representing the most recent data in the previous 24-hour period, two hidden layers of sizes 72 and 48, and an output layer of size 24, representing 24 steps ahead forecasts. The *tanh* activation function is used in all the layers, and the maximum number of epochs is set to 400. If during the training process, the training SSE continues to increase 8 consecutive times, then the training is terminated early and the parameters are reverted back to the model that produces minimal SSE. The *tanh* function also dictates that the data be normalized since it can only output values from -1 to 1. The training data are normalized in between -0.8 to 0.8 using Min-Max Normalization in order to leave room for the testing data to contain values greater than or less than the maximum and minimum values in the training set, respectively. Other important parameters include the learning rate, size of the mini-batch and the regularization parameter are found using a grid search on a small sample of data. The learning rate is then set to 0.1, the mini-batch size is 50 and the regularization parameter is set to 0.5.

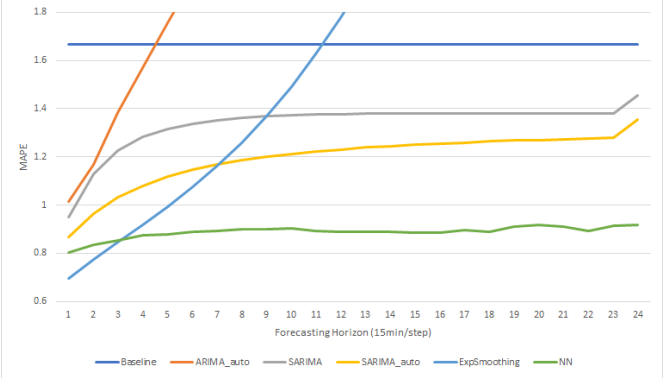

Fig. 6.9: Performance Comparison using MAPE

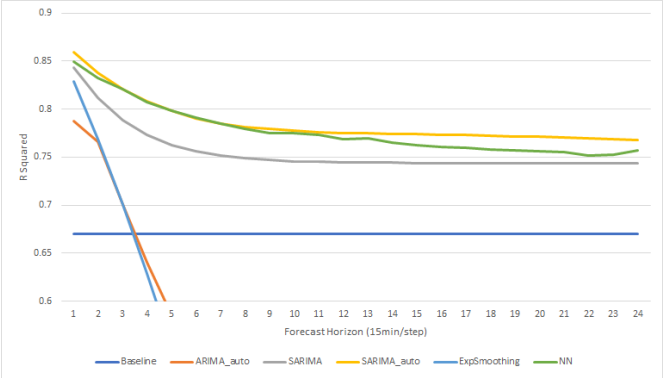

Fig. 6.10: Performance Comparison using R^2

Rolling forecasts are used, in each iteration 12 weeks of data are used for training and forecasts are produced on 8 weeks of testing data, 24 steps ahead at a time. The metrics of evaluation include Mean Absolute Percentage Error (MAPE) and Coefficient of Determination (R^2). No imputed values are included in computing the evaluation metrics. An additional baseline, which is the historical average of the same weekdays at a particular time in the training set, is included for comparison purposes.

In both Figure 6.9 and 6.10, ARIMA_auto and SARIMA_auto are the models that used automated search in order to find the appropriate orders that optimizes the AICc criterion. In terms of MAPE, as shown in Figure 6.9, Neural Networks performed the best overall, and the overall rate of performance degradation seems low. The SARIMA_auto performed consistently better than the SARIMA$(1, 0, 1) \times (0, 1, 1)$ model, but both exhibit similar trends. The Exponential Smoothing model performed better than the ARIMA_auto model, but both models only produced satisfactory forecasts in the short terms. The performance degradation rates for both models seem rather high. In terms of R^2, the relative performance of most models have stayed the same, with the exception of SARIMA_auto, which seems to be competitive with Neural Networks.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this paper, we presented the ScalaTion TSDB, an effective and efficient open-source TSDB. Our experiments showed that ScalaTion TSDB is competitive with existing state-of-the-art TSDBs for many of the database operations; for certain operations such as aggregation, projection, select, intersect and union, the ScalaTion TSDB is significantly faster than its competitors. ScalaTion TSDB provides easy integration with a great variety of existing forecasting models provided by the ScalaTion analytics framework without the need to rely on external libraries. We have demonstrated ScalaTion TSDB with one case study; it can be extended to support other domains too. Three easy-to-use Application Programming Interfaces, namely Algebraic Interface, SQL-like Interface, and Functional Interface, are provided in ScalaTion TSDB to accommodate a diverse group of users who may freely choose their preferred API. We discussed factors which determine compression ratio and analytics performance. ScalaTion's variety of forecasting techniques can be benefited when combined with the spatial-temporal features. We provide an easy to build/ install framework with well documented feature rich libraries which can help future researchers and developers to contribute to their research.

In the future, we plan to introduce temporal indexing to make our algorithms more efficient. Our TSDB can be extended to implement a spatial-temporal database by adding spatial attributes. Spatial functions, spatial aggregate functions, spatial operators can also be implemented. In the future, with the help of spatial information combined with temporal information, more insights can be gained and more accurate time series analysis can be performed, for example, we may achieve better accuracy in forecasting traffic when precipitation data, traffic direction, traffic speed, concert events, road accidents, road repair work, etc. are considered. Dynamic time warping can be implemented to align 2 irregular time-series.

The JVM does not support vector instructions set as of this writing and JNI calls to native C code are expensive. We have provided a vectorized algorithm for matrix-matrix multiplication in Scala; since similar auto-vectorized implementation in native C has shown significant performance gain. Various Linear Algebra operations can be implemented on similar lines. Vectorized implementation of such operations is expected to improve the performance of neural networks in ScalaTion. Currently, our block matrix multiplication algorithm works for square matrix sizes with perfect square dimensions. Work can be done in this area to make it available for all sizes of matrices. SUMMA matrix multiplication algorithm does not have a 1D array version. However, as can be seen from the performance evaluations section and the tables in the Appendix section, 1D arrays performs better than their 2D versions. Work can also be done in making 1D implementation of SUMMA parallel.

Bibliography

[1] D. Schmidt, A. K. Dittrich, W. Dreyer, and R. Marti, "Time series, a neglected issue in temporal database research?" in *Recent Advances in Temporal Databases*. Springer, 1995, pp. 214–232.

[2] R. T. Snodgrass, "Temporal databases," in *IEEE computer*. Citeseer, 1986.

[3] L. Deri, S. Mainardi, and F. Fusco, "tsdb: A compressed database for time series," in *International Workshop on Traffic Monitoring and Analysis*. Springer, 2012, pp. 143–156.

[4] X. Zhao, "High performance algorithms for multiple streaming time series," Ph.D. dissertation, New York University, Graduate School of Arts and Science, 2006.

[5] J. Shieh and E. Keogh, "*i*SAX: indexing and mining terabyte sized time series," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2008, pp. 623–631.

[6] Dan Worth, "Internet of Things to generate 400 zettabytes of data by 2018." [Online]. Available: https://www.v3.co.uk/v3-uk/news/2379626/internet-of-things-to-generate-400-zettabytes-of-data-by-2018

[7] J. A. Miller, H. Peng, and C. N. Bowman, "Advanced tutorial on microscopic discrete-event traffic simulation," in *Simulation Conference (WSC), 2017 Winter*. IEEE, 2017, pp. 705–719.

[8] H. Peng, S. U. Bobade, M. E. Cotterell, and J. A. Miller, "Forecasting traffic flow: Short term, long term, and when it rains," in *International Conference on Big Data*. Springer, 2018, pp. 57–71.

[9] V. J. Tsotras and A. Kumar, "Temporal database bibliography update," *Sigmod Record*, vol. 25, no. 1, pp. 41–51.

[10] C. M. Saracco, M. Nicola, and L. Gandhi, "A matter of time: Temporal data management in DB2 10."

[11] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala, "Temporal query processing in Teradata," in *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013, pp. 573–578.

[12] Jonathan S. Katz, "Range types: Your life will never be the same," 2012, accessed: 2018-07-28. [Online]. Available: https://wiki.postgresql.org/images/7/73/Range-types-pgopen-2012.pdf

[13] N. Leavitt, "Will NoSQL databases live up to their promise?" accessed: 2018-07-28. [Online]. Available: http://www.leavcom.com/pdf/NoSQL.pdf

[14] A. A. Ouassarah, "Adi : A nosql system for bi-temporal databases," Ph.D. dissertation, Universit de Lyon, 2016.

[15] M. Kaveh, "ETL and analysis of iot data using OpenTSDB, Kafka, and Spark," Master's thesis, University of Stavanger, 2015.

[16] A. Bader, O. Kopp, and M. Falkenthal, "Survey and comparison of open source time series databases," *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*, 2017.

[17] "Spatiotemporal databases - wiki," accessed: 2018-07-28. [Online]. Available: https://en.wikipedia.org/wiki/Spatiotemporal_database

[18] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proceedings of*

the VLDB Endowment, vol. 8, no. 12, pp. 1816–1827, 2015.

[19] "R project - time series analysis," accessed: 2018-07-21. [Online]. Available: https://cran.r-project.org/web/views/TimeSeries.html

[20] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with Python," in *9th Python in Science Conference*, 2010.

[21] Y. Fang, "Analytics databases: A comparative study," Master's thesis, University of Georgia, Franklin College of Arts and Sciences, 2018.

[22] Oracle Javadoc Contributors, "Java platform, standard edition 8 API specification." [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html

[23] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso, "The database-is-the-service pattern for microservice architectures," in *International Conference on Information Technology in Bio-and Medical Informatics*. Springer, 2016, pp. 223–233.

[24] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 671–682.

[25] V. G. Harish, V. K. Bingi, and J. A. Miller, "A big data platform integrating compressed linear algebra with columnar databases," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 2344–2352.

[26] K. Goto and R. van De Geijn, "On reducing TLB misses in matrix multiplication," Technical Report TR02-55, Department of Computer Sciences, U. of Texas at Austin, Tech. Rep., 2002.

[27] "SUMMA: Scalable universal matrix multiplication algorithm, author=Van De Geijn, Robert A and Watts, Jerrell, journal=Concurrency: Practice and Experience, volume=9,

number=4, pages=255–274, year=1997, publisher=Wiley Online Library."

[28] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, p. 14, 2015.

[29] J. Nie, B. Cheng, S. Li, L. Wang, and X.-F. Li, "Vectorization for Java," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2010, pp. 3–17.

[30] OpenJDK Contributors, "Vectors for java," 2018. [Online]. Available: http://cr.openjdk.java.net/~vlivanov/panama/vectors/vectors.html

[31] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015.

[32] W. Shi, "Heterogeneous distributed shared memory on wide area network," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 71–80, 2001.

[33] "Apache spark - issues tracker," accessed: 2018-07-28. [Online]. Available: https://issues.apache.org/jira/browse/SPARK-5293

[34] "Apache spark - github commit log," accessed: 2018-07-28. [Online]. Available: https://github.com/apache/spark/commit/bc1babd63da4ee56e6d371eb24805a5d714e8295

[35] "Akka documentation - cluster usage," accessed: 2018-07-28. [Online]. Available: https://doc.akka.io/docs/akka/2.5/cluster-usage.html

[36] Glenn Engstran, "Functional programming and big data," 2014. [Online]. Available: http://glennengstrand.info/media/fpbd.pdf

[37] P. Chiusano and R. Bjarnason, *Functional programming in Scala.* Manning, 2015.

[38] J. C. Molloy, "The open knowledge foundation: open data means better science," *PLoS biology*, vol. 9, no. 12, p. e1001195, 2011.

[39] "Time series for spark (the spark-ts package)," accessed: 2018-08-01. [Online]. Available: https://github.com/sryza/spark-timeseries

[40] "Flint: A time series library for apache spark," accessed: 2018-08-01. [Online]. Available: https://github.com/twosigma/flint

[41] M. Lippi, M. Bertini, and P. Frasconi, "Short-Term Traffic Flow Forecasting: An Experimental Comparison of Time-Series Analysis and Supervised Learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 2, pp. 871–882, 2013.

[42] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic Flow Prediction with Big Data: A Deep Learning Approach," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 865–873, 2015.

[43] G. E. Box and G. M. Jenkins, "Time Series Analysis Forecasting and Control," DTIC Document, Tech. Rep., 1970.

[44] C. Holt Charles, "Forecasting trends and seasonal by exponentially weighted averages," *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 1957.

[45] P. R. Winters, "Forecasting Sales by Exponentially Weighted Moving Averages," *Management science*, vol. 6, no. 3, pp. 324–342, 1960.

[46] R. J. Hyndman, Y. Khandakar *et al.*, *Automatic time series forecasting: the forecast package for R.* Monash University, Department of Econometrics and Business Statistics, 2007, no. 6/07.

[47] S. Shekhar and B. Williams, "Adaptive Seasonal Time Series Models for Forecasting Short-Term Traffic Flow," *Transportation Research Record: Journal of the Transportation Research Board*, no. 2024, pp. 116–125, 2008.

[48] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2014.

APPENDIX

For the TABLE 7.1, 7.2, and 7.3 below, times are given in ms.

Table 7.1: Matrix Multiplication results for 900 * 900 matrix size

Method	**2DS**	**1DS**	**2DP**	**1DP**
Generic	37474	NA	3648	NA
Naïve	3074	1679	315	87
Optimized	1210	1170	75	65
SUMMA	1796	NA	167	NA
Naïve Block	1210	1229	92	58
blis through JNI	NA	424	NA	NA

Table 7.2: Matrix Multiplication results for 1800 * 1800 matrix size

Method	**2DS**	**1DS**	**2DP**	**1DP**
Generic	164302	NA	27376	NA
Naïve	35590	15380	6892	1808
Optimized	7370	7151	1740	1266
SUMMA	5617	NA	289	NA
Naïve Block	6882	7263	1170	978
blis through JNI	NA	982	NA	NA

Table 7.3: Matrix Multiplication results for 2500 * 2500 matrix size

Method	**2DS**	**1DS**	**2DP**	**1DP**
Generic	626852	NA	123601	NA
Naïve	135366	51662	23638	9647
Optimized	32240	26268	6276	7460
SUMMA	22243	NA	2658	NA
Naïve Block	35334	27940	6870	4082
blis through JNI	NA	2618	NA	NA

Table 7.4: Supported Times Series Analysis Techniques

Name	Package	Class
Exponential Smoothing	forecaster	ExpSmoothing
Auto-Regressive	forecaster	AR
Moving-Average	forecaster	MA
ARMA	forecaster	ARMA
ARIMA	forecaster	ARIMA
SARIMA	forecaster	SARIMA
SARIMAX	forecaster	SARIMAX
Dynamic Regression	forecaster	DynRegression
Functional Smoothing	fda	Smoothing_F
Functional Regression	fda	Regression_F
Fourier Transform	calculus	FFT
Wavelets	calculus	Wavelet
Neural Networks	analytics	NeuralNet_3L
Neural Networks	analytics	NeuralNet_XL
Recurrent Neural Nets	analytics	RNN_Net
Long Short-Term Memory Nets	analytics	LSTM_Net
Convolution Neural Nets	analytics	CNN_Net
Temporal Convolution Nets	analytics	TCN_Net