Analytics Databases: A Comparative Study

by

Yang Fan

(Under the Direction of John A.Miller)

Abstract

With the emergence of the Big Data era, high performance analytics databases are highly in need in areas such as business intelligence and predictive analytics. Column-oriented databases are created as a type of NoSQL (Not only SQL) databases to fulfill those needs. ScalaTion is an open-source Scala based tool for simulation, optimization and analytics, and it includes an implementation of column-oriented in-memory database that can handle high performance analytics. The database provides an easy way to transform a table into a matrix which may be used as input for other advanced machine-learning models that are also available in ScalaTion. Fifteen different experiments are conducted to evaluate the performances of five databases: ScalaTion, MySQL, SQLite, SparkSQL and ClickHouse. The performance of ScalaTion is for the most part on par with those of open-source column-oriented databases and at times can be significantly better.

Index words: Analytics database, Column-oriented database, Big Data analytics pipeline

Analytics Databases: A Comparative Study

by

Yang Fan

B.Eng., Shenzhen University, 2013

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

Analytics Databases: A Comparative Study

by

Yang Fan

Acknowledgments

A special thanks to my major professor Dr. Miller for all his patience and guide through my research journey. The knowledge and guidance he shares with me is tremendous. I would like to thank Dr Krzysztof J. Kochut for being a part of my committee and I learnt a lot from his course Software Engineering. I would also like to thank Dr Shannon Quinn for accepting to be a part of my committee and giving me knowledge about big data analytics. I also thank Supriya Ramireddy, Vinay Kumar Bingi, Santosh Uttam Bobade, Hao Peng for their help and company during my thesis journey. I would like to thank friends and family for all their unconditional support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The “relational database” term was invented by E. F. Codd in 1970. Later this term was officially defined in his paper [1]. SystemR [2], Ingres Database [3], Oracle Database were built upon this model during 1970 and 1980. Oracle released the first commercial RDBMS in 1979 [4]. Since then, relational database management systems (RDBMS) have been regarded as a mainstream database system. There is a domain-specific language Sequel [5] which came along with RDBMS. It helps users to manage data in RDBMS. Due to its convenience, SQL (Sequel) is familiar to all professional and other users. It has been a predominant way to manage data in database management system. RDMBS is known to be good for handling transaction processing. When creating a big database system, data architects and data modelers often need to have a clear and complete understanding of their data models. Every table needs to have its own schema indicating the type and name of each column. RDMBS typically supports ACID (Atomicity, Consistency, Isolation, Durability) transactions. With the emergence of the Big Data era, data started growing exponentially in different forms. This results in frequent table modifications and RDBMS, by its design, shows its weakness in catering to these needs. RDBMS is unable to react agilely to table modification and table creation.

New types of database systems which provide more flexibility and have better performance in analysis have been discussed in the industry and open-source community. These types of database systems are called as NoSQL databases. They are different from the traditional database systems in many ways. Papers [6, 7, 8] have discussed the features of NoSQL databases in detail from different perspectives. There are no policies to govern data centrally in these databases [6]. These databases provide various different ways to store data of large sizes, support nested data structure and data analysis with great performance, etc. When designing a NoSQL database, application architects and developers will design the database systems based on the queries the application will use. Theses systems to some extent shift the tasks of being ACID to the users and application programmers. On the other hand, these kinds of databases release users from making complicated schema and at the same time provide high availability and low latency.

There are several categories of NoSQL database systems, including graph databases, key-value databases, document databases, and column-oriented databases. Relational data model represents data as n-ary relation R which has a set of distinct tuples and every column has a name with a specific domain. Data model in graph databases uses a set of vertex, a set of directed edges, a set of vertex labels to present data. Some have a set of vertex properties or a set of edge properties which are used to distinguish subset of vertex or edges. Document databases store all information for a given object in a single instance. They store an additional metadata file to support other feature such as organizing documents and execution engine optimization. Column-oriented databases completely vertically partition a database into a collection of individual columns that are stored separately. Some data model consists of different projections of table and some are represented as n segmentations that each segmentation is stored separately in different mechanisms. These segmentations are reconstructed back to a tuple using a virtual ID (usually the position of the tuple). Some of these systems support ACID transactions and some support BASE (Basically Available, Soft state, Eventual consistency) transactions. All of these types have gained a certain amount of popularity in open source community and industry.

Companies are facing the problem that huge amount of data are created and introduced into their system everyday. They need to have a better, easier and cheaper way to store the data and to handle some real-time operations on them. And there is no existing RDBMS that can solve their problems completely. Therefore, many software product companies have built their own NoSQL database system according to their specific own needs. Google has Bigtable [9]. Facebook has Cassandra [10] and LinkedIn has Voldemort [11]. Column-oriented databases as one type among these NoSQL database systems have attracted great attention. The development of e-Commerce and business intelligence brings a huge request for fast real-time analysis. How to retrieve a value from a huge amount of data in a short time or doing analysis on these values as part of the report or application information has been the original task of column-oriented database.

In terms of storage, column-oriented database has three ways to stores columns: on-disk, in memory, and hybrid. They have different advantages, disadvantages and trade-offs. To achieve persistence, the database system needs to store the data on the disk or on NVDIMM (non-volatile dual in-line memory module). For column-oriented database, values in a column are often repetitive. Compression algorithms can be applied to each column. After compression, systems only need to store a representative value for each repeated value, saving many space. Some of these systems designed their own data structure to store the columns, which allow operations to be done without decompressing the data. This greatly improves the performance of all kinds of operations. Another aspect, different from how row-oriented databases retrieve data as a whole tuple, column-oriented databases only need to retrieve related columns, reducing many seeking when reading from disk. In-memory types of column-oriented databases will have similar benefits when bring data from memory to cache. Large data which does not fit in the memory can be solved with a distributed mode of these data systems. Most of the time, on-disk column-oriented database system have their own way and related algorithms to encode and store the column on disk. They store side information to reconstruct the data back into the table and have optimizers to organize the query executions which can reduce IO by filtering out useless values in columns. A good mechanism for bringing the data from disk to memory and a decision of frequency and timing to cache useful values makes on-disk column-oriented databases system competitive in speed with in-memory column-oriented database. Column-oriented databases have been known to be good for read oriented applications but clumsy for writing and updating [12]. Many NoSQL databases only support delete and insert instead of update operations to reduce concurrency problems. Some of NoSQL databases do not support join. However, study shows, in column-oriented databases, join operations can be leveraged using indexes before getting the values from disk [13]. C-store[14] supports joins in a competitive way with row-oriented relational database join operations. Column-oriented databases also have advantage to do analytics processing. Despite the weakness in updating and adding values to a column-oriented database, aggregation functions show a strength when compared to row-oriented database [6].

Vertical scaling is achieved by adding more cores (ect Intel now supports at most 22 cores) and horizontal scaling is achieved by adding more machines, which is a process of sharding the data. Horizontal scaling is usually automatically done in NoSQL databases. ClickHouse [15] can process queries in parallel. In terms of query language, ClickHouse [15] and C-Store [14] support SQL query.

Druid is an open source shared-nothing, distributed system which has column-oriented persistence storage [16]. Different nodes of different types are responsible for different tasks. These nodes work cooperatively to solve complicated analytics problems. Druid uses bitmap sets to leverage the filtering of rows. Bitmap is a method which uses a list of bits to represent the filter result of each column. Bit 1 indicates the value in the column returns true for the filter predicate, and vice versa. Later they perform and or or operations on the lists of bits from different columns and map to related rows to get the filter result. They also apply compression on column data to save space when storing on disk or in memory. C-Store [14] is another open source tool, released in October, 2006. The system has two separate stores architecture. One is Read-Optimized Store and the other is Writable Store. The two parts are connected using Tupler Mover. The Read-Optimize store is comprised of different projections of the original table. The cooperation of these two stores helps C-store to be fast in read and relatively fast in update. Multiple copies of different projections stored in different sorting orders also make C-store possess high availability. SAP HANA [17] is hybrid storage, on-disk data management system. SAP HANA has different strong query process engines which can handle different types of data structures to support high performance analytics for different types of data. It supports SQL-script and has a query optimizer. Another highlight of SAP HANA is supporting ACID which some column-oriented database systems do not do.

Based on the experiences and knowledge with existing column-oriented databases, our column-oriented database system in ScalaTion is now capable of performing a relatively fast load and other common relational algebra operations. Users can load a CSV (Comma Separated Values) file into the database or simply add the raw data as a Sequence of Vectors into the database as Relation. ScalaTion has a group of fundamental linear algebra classes such as VectorD, which have fast search, select, append, etc. operations. The database columns are stored in these Vectors. Later functions are built on these Vector functions and combined using index.

To solve the disjointing of data process and data modeling and to better combine Big Data analysis and relational algebra processing, we built RelationFrame API. RelationFrame shares the same data structure with Relation. It provides high order functions such as filter, map, fold, and reduce. So users can manipulate data in condense codes, finishing complex calculations.

The rest of this document is organized as follows: Background and goals of Relation and RelationFrame are presented in Chapter 2. More detail about the programming interface is shown in Chapter 3. In Chapter 4, we present performance evaluations between on-disk RDBMS, in-memory RDBMS, on-disk column-oriented database, column-oriented database and ScalaTion. We present related work in Chapter 5. Conclusions and future work are discussed in Chapter 6.

Chapter 2

Background and Goals

2.1 ScalaTion Overview

ScalaTion is an open-source Scala [18] based tool for simulation, optimization and analytics. It was first released in 2010, under the MIT License. ScalaTion is built upon a number of high-performance stable linear algebra classes and has rich data science/machine learning algorithm libraries. The linear algebra library consists of different types of Vector, Sparse Vector, RleVector[19], Matrix, Sparse Matrix, BidMatrix, SymTriMatrix classes. These linear algebra classes provide a solid foundation for ScalaTion to handle heterogeneous data and support high performance operations. ScalaTion also provides parallel versions of most of these operations. ScalaTion needs to handle many analytics and column-oriented systems have strength in analytics as mentioned in Chapter 1. Therefore, ScalaTion has a in-memory column-oriented database system implemented. The database supports data storage and relational algebra operations. The columns are stored separately and are in terms of different types of Vector, which have fundamental optimized operations, such as append, filter, and slice.

The modeling in ScalaTion uses Matrix and Vector as the type of inputs. After loading the data from CSV into Relation, toMatriD or toVetorD functions can be called to transform the Relation into linear algebra classes. Here is an example to show how to load a CSV file into a Relation and transform it into either a matrix or a vector using ScalaTion.

```
// null indicates the domain is not given
// 0 indicates the primay key is the first column
val testRelation = Relation(path, "testRelation", null, 0)
```

```
// from column 1 to column 1000 are the attributes
val  x = testRelation.toMatriD(1 to 1000)
// column "label" is the label column
val y = testRelation.toVectorD("label")
```

These codes create a testRelation Relation by reading a CSV file from path and use toMatriD() method to create a MatriD[1] consisting of data from column1 to column1000 as input x. These codes later use toVectorD() method to create a related VectorD using column "label" as input y.

2.2 Addtions to Relation

Previously, ScalaTion provided basic relational query processing such as select, project, intersect, union. From the perspective of data processing, many data processing operations were not supported in Relation. So ScalaTion users need to use other language such as R [20] or Python [21] to operate on their raw data before loading the data into ScalaTion modeling. We enrich the library by adding more complicated operations such as aggregation, groupby, where and join.

2.3 Goals for RelationFrame

With the rich and solid supports of the linear algebra library in ScalaTion, we wish to enrich our Relation API with more advanced operations. We also want to build a new RelationFrame API inside ScalaTion which offers clear and user-friendly ways to manipulate data at a higher level. And both of Relation and RelationFrame can be quickly and easily transformed to Matrix and Vector as input to ScalaTion modeling and simulation.

We set the following goals for Relation API and RelationFrame API:

1. Support more advanced relational processing within ScalaTion.
2. Provide high performance using established DBMS techniques.
3. Easy interface for complex operations.

[1]A matrix of type Double

4. Support user-defined operations on the RelationFrame.

5. Parallelism should be applied.

Chapter 3

Programming Interface

Relation and RelationFrame runs as a library as a part of ScalaTion, as shown in Figure 3.1. We first introduce Relation API and later cover RelationFrame API, which allows users to do functional processing and relational processing, alternately. User defined functions are supported in RelationFrame. We will be discuss UDF (user-defined functions) in Section 3.6.

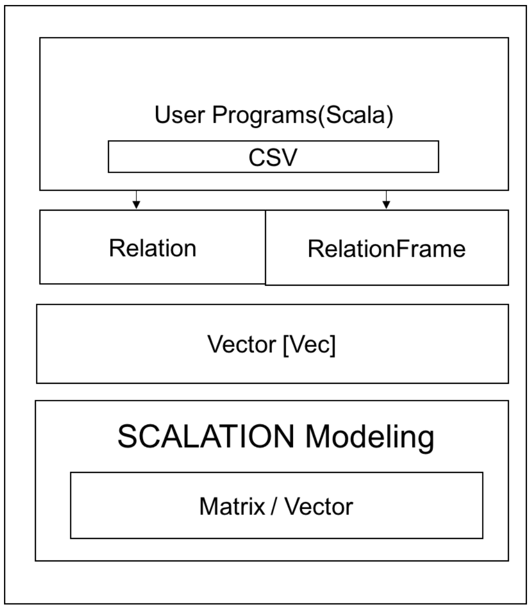

Figure 3.1: Relation and RelationFrame and their interaction with ScalaTion

3.1 Relation API

Relation is a Vector of Vec [1] in abstract. It is analogous to a table in relational database. It has simple relational query processing functions such as select, project, union, intersect, join. Extended functions are groupby, aggregate, where.

A simple example to illustrate a basic analytic pipeline using ScalaTion is as below. The Scala code below defines a Relation from a CSV file and performs groupby and aggregation on the Relation.

```
// null indicates the domain is not given
//0 indicates the primay key is the first column
val testRelation = Relation(path, "testRelation", null, 0)
val result = testRelation.groupBy("CITY")
                .epi(Seq(max), Seq("ZIP"), "UNITID", "CITY")
```

In this code, Relation will be created using the CSV file in path. Groupby method will group the relation by the CITY column, store the grouping information in a Seq structure and sort the rows based on the groupby attributes. Later the epi function will calculate the max of ZIP for each group and project on UNITID, CITY columns and the aggregate column.

Compared with SparkSQL, SparkSQL performs the task in the following way:

```
val testRelation = spark.read
      .format("com.databricks.spark.csv")
      .option("header", "true") // Use first line of all files as header
      .option("inferSchema", "true") // Automatically infer data types
      .load(path)
val data_max = testRelation.groupBy("CITY")
                            .max("ZIP").alias("counts")
val result = testRelation.select("UNITID", "CITY")
                  .join(data_max, "CITY").orderBy(testRelation("CITY"))
```

[1] Vec is a trait, it establishes a common base type for all vectors

3.2 RelationFrame API

RelationFrame is an extension of Relation. It provides functions to manipulate on data at higher level. It can be constructed from Relation and later filled into ScalaTion modeling. The following code show how to create a RelationFrame from a Relation.

```
val relationFrame = RelationF(testRelation)
```

3.3 Data Model

Relation is a Vector of Vec. Vec is a trait which establishes a common base type for all different types of Vectors in ScalaTion linear algebra package. It supports various primitive types Int, Double, Long and some specially defined types. There are 4 advanced types defined in ScalaTion which can be used to describe a data. They are Complex, Real, StrNum, Rational. These advanced types are specially defined to handle different kinds of data in a more specific way. Table 3.1 will introduce usage and constructor of these types in detail.

3.4 Relation Operations

The library is enriched with more operations such as aggregate, groupby and join. Indexes are added to the system. Index join, which uses indexes, is implemented. Adding indexes speeds up some operations and allows for storing information as sequence of indexes instead of rebuilding the relation. However, maintaining indexes burdens some other operations. So a good balance of maintaining indexes and skipping the maintenance is critical to performance. When the operation, such as project, is applied, the table will be changed vertically. The indexes which are created when the table is created are no longer valid. If maintenance is necessary, decisions of when to rebuild indexes avoiding unnecessary rebuilding also need to be optimized.

Type Name	**Usage**	**Constructor**
Int	Represents integer number with 32-bits	val num = 100
Long	Represents integer number with 64-bits	val num = 9223372036854775807L
Double	Represents decimal number with 64-bits (supports 53 bits of precision)	val num = 0.25
Complex	Represents and operates on complex numbers Example: $\sqrt{-1} = i$	construct 2.0+3.0i val num = Complex(2.0, 3.0)
Real	Complex number is consisted of a real number and an imaginary number Provides higher precision floating point numbers (supports 106 bits of precision)	val num = Real(0.25)
Rational	Represents and operate on rational numbers	construct $\frac{1}{4}$ val num = Rational(1l, 4l)
StrNum	Represents and operate on string numbers	val num = StrNum("0.25")

Table 3.1: Explanation of Different Types in ScalaTion

Vec is immutable in ScalaTion. Constructing Relation become costly because of this. A temporary column which is a ReArray[2] is used to help construct a Relation.

The groupby function is just an intermediate process of aggregation. Table creation and updating in column-oriented databases is expensive. It is unnecessary to create a new relation out of that. To save space and time, two variables are used to store the information created by groupby. One is *OrderedIndex* and the other is *Grouplist. OrderedIndex* as the name indicates, it is used to store the ordered indexes. After the groupby operation, aggregation projection needs the rows to be sorted based on the groupby attributes. The order of indexes are stored

[2]The 'ReArray' is a class in ScaLation which provides an implementation of mutable, resizable/dynamic arrays.

in the *OrderedIndex*. For each group, the index of the first row is stored in *Grouplist*. So later, group list information can be retrieved from *Grouplist*.

The aggregation operation has two main tasks to do. One is to calculate the aggregation and the other is to project on specific columns. Aggregation is calculated by calling the aggregateFunction for each aggregate columns. When the projected columns have no one-to-one relation with group by attributes, only the first tuple of each group will be shown in the result.

Epi function in ScalaTion is an API which are used to do the aggregation and projection. A list of aggregate function, a list of aggregate columns and a list of project columns are given as input of epi function. An example of epi API is shown as following.

```
\\ max on ZIP and project on UNITID, CITY and max columns
relation.epi(Seq(max2), Seq("ZIP"), "UNITID", "CITY")
```

The epi function combine project and aggregate in one API. When one of the project columns does not have a one-to-one relationship with groupby columns, MySQL does not support the query execution when “ONLY_FULL_GROUP_BY SQL” mode is enabled(which is by default). When the “ONLY_FULL_GROUP_BY SQL” mode is disabled, a random tuple from each group of projected columns will be selected as result (later referenced as easy version query). A full result set can be created by using join in MySQL (later referenced as complicated version query). Epi is used for the complicated version. Another API EpiEasy is used for the simple version.

ScalaTion now supports index join. After comparison, implementation with HashMap performs better than the one with TreeMap for our system. A HashMap is used to hash a primary key to every tuple. The index join only works when the join predicate is on the primary key of either or both of the two tables which is a common case in query processing. Figure 3.2 shows performance of the nested loop join, the index join with HashMap, the index join with TreeMap and the index join with BpTree. BpTree though performs the worst for index join among these 3 indexes. Range select operation will be optimized using

BpTree index. Execution time of larger testing datasets of nested loop joins is too large to be recorded.

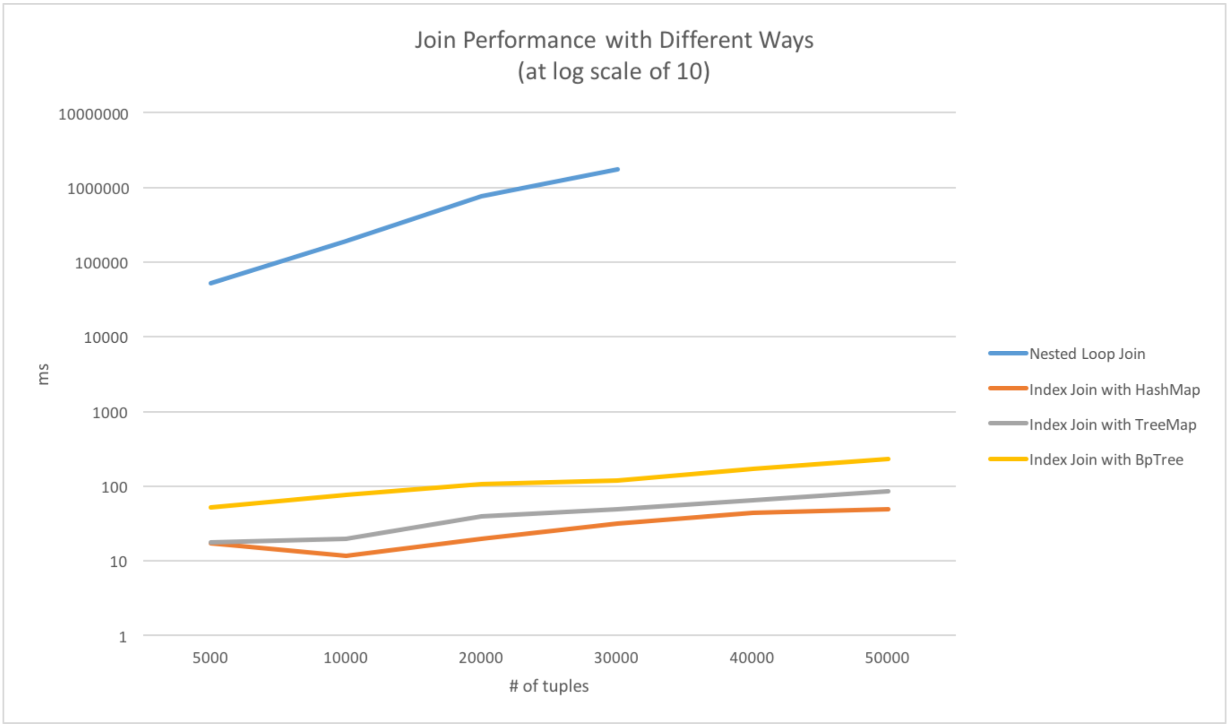

Figure 3.2: Join Performance with Different Implementation

3.5 RelationFrame Operation

Data frames refer to a data structure representing cases (rows) and each of which consists of a number of observations or measurements (columns) [3]. Tools with similar functions are R DataFrame [20], Spark DataFrame [24] and Python DataFrame [21]. In ScalaTion, users can perform relational operations and high order functions on RelationFrame. RelationFrame supports all common relational operators, including projection (select), filter (where), join, and aggregations (groupby, epi). And it also has high order functions such as filter, map, reduce and fold. These operators take specific inputs and translate the information to compute results.

[3]https://github.com/mobileink/data.frame/wiki/%What-is-a-Data-Frame%3F

The following code shows how to filter on two different tables on the Date column which has value of 20180101 and join the two tables together by using UNITID.

```
val newfuture = futuretable.filter( _ == "20180101", "Date")
val newcurrent = currenttable.filter( _ == "20180101", "Date")
val fulltable = newcurrent.joinindex("UNITID", newfuture)
```

Same results are created using SparkSQL as followed:

```
val newfuture = futuretable.filter(futuretable("Date") === (20180101))
val newcurrent = currenttable.filter(currenttable("Date") === (20180101))
val fulltable = newcurrent.join(newfuture, "UNITID")
```

Pid	**Name**	**Years**
01	Jack Johnson	5
02	Adam Oldman	1
03	Bill Peeler	6
04	Ken Smiths	2
05	James Lee	1

Table 3.2: Table Professor

Following sections will use a professor table to show some high order functions examples using RelationFrame. The table of professor is shown in Table 3.2.

Here is an example shows how to add 1 to every value of Years column using Relation-Frame:

```
  val result =  professor.map((x: Int) => x+1,"Years")
```

Years
06
02
07
03
02

Table 3.3: Result Table after Map on Professor

The result of this code is shown in Table 3.3.

Same results can be created from this code using SparkSQL:

```
val result = df.map
{case Row (pid: Int, name: String, Years:Int) => (Years + 1)}
```

The following code shows how to get the sum of all the values of Years column:

```
professor.reduce((x:Int, y:Int) => x+y,"Years")
\\result: 15
```

Same results will be created from this code using SparkSQL:

```
professor.select("Years").reduce((x,y) => Row(x.getInt(0) + y.getInt(0)))
```

The following code show how to get the sum of all the values of Years column with a default starting sum, 10:

```
def plus(x:Int, y:Int): Int =  x+y
professor.fold(10, plus, "Years")
\\result: 25
```

Same results can be created from these codes using SparkSQL:

```
val result = professor.select("Years")
                    .reduce((x,y) => Row(x.getInt(0) + y.getInt(0))) + 10
```

3.6 RelationFrame versus Relational Query Languages

RelationFrame can be easily transformed into Matrix and Vector to fill into data science machine learning modeling algorithm, without saving the intermediate results. It provides clear and user friendly API and convenient way to finish some common data processing operations.

The following example shows how to create a new statistic column which is max of a column of a specific group and later fill in a regression model in ScalaTion:

```
val student = Relation(path, "student", null, 0)
val result = student.groupBy("SCHOOL")
           .epi(Seq(max), Seq("GRADE"), "ADDR", "CITY")
val rg = new Regression
              (result.toMatriD(1 to 3), result.toVectorD("Label"), Cholesky)
             // use Cholesky Factorization
```

Relational query produces same result using MySQL inside R:

```
install.packages("sqldf")
library(sqldf)
rs <- dbSendQuery(con, "select ADDR, CITY,
                    max(GRADE) as max FROM student group by SCHOOL")
student <- fetch(rs, n=-1)
```

fit <- lm(LABEL ~ ADDR + CITY + max, data=student)

3.7 User-Defined Functions

To define user-defined functions in MySQL, you need to install object files in addition to the server itself [4]. RelationFrame supports inline definition of UDFs without complicated packaging and registration process. In RelationFrame, UDFs can be passed by passing Scala functions. This functionality provides users customized way to process their data inside ScalaTion. For example, multi function defines a function multiplying two numbers. Given this function as an argument to map function of RelationFrame, the Years column will be applied to this multi function and return a new column:

```
def multi (x:Int): Int =  x*10
professor.map (multi, "Years").show()
```

Here multi is a user defined function which takes an integer and multiply it by 10. The multi function is given as an input for the map function. In the map function, every value of Years column will be multiplied by 10. A new column will be returned.

The result shows in Table 3.4

[4] https://dev.MySQL.com/doc/refman/5.7/en/adding-functions.html

Years
50
10
60
20
10

Table 3.4: Table Professor After Map Operation

Chapter 4

Performance Evaluations

4.1 Ralational Operations Dataset and Join Dataset

Data of this study was released in October, 2015 by College ScoreCard under the United States Department of Education (https://collegescorecard.ed.gov/data/). This dataset has attributes representing a student's information about his school, the city of the school and some other unknown personal information. Each row in the data stands for a student cohort admitted to a certain university.

We use the first 500 columns of the tables to form a relational operations dataset as do our relational operations experiments and dataframe experiments. We make different sets of dataset whose rows size differs from 1000 to 90000.

For the join operation testing, we take the first 10 attributes and split them into two datasets. One consists of 4 attributes and the other consists of 6 attributes. We added Sid as key to present the student id of whom this information belongs to. The two tables are named student and address in the database.

4.2 Pipeline Dataset

The original raw data has total 1729 columns. Based on the researcher's study result [22] 31 columns are chosen through feature selection processing and data from 2001, 2003, 2005, 2007, 2009, 2011 are chosen as training set. We replace the null and empty value with 0. After filtering and join operation, total count of rows is 17813. These 17813 data are used to do regression.

These experiments were conducted on a system with an Intel Core i5 2.70 GHz processor running the 64-bit OS X High Sierra 10.13.2 distribution of the Mac operating system with 8 GB of memory. The versions of SparkSQL, MySQL, SQLite, Python and R are 2.11.7, 6.3.4, 3.22.0, 2.7 and 3.4.3, respectively. Spark is set up to run in one thread. ClickHouse is deployed on a micro t2 instance of EC2. The virtual machine has 1GB of memory, 1 cpu, 1 core and is running Ubuntu operating system.

4.3 Relational Algebra Performance

The following eight sections are showing relational processing performance comparison results of five different database management systems. We choose an on-disk relational database, MySQL, an in-memory relational database SQLite, an on-disk column-oriented database ClickHouse, and an in-memory row-oriented dataframe SparkSQL.

4.4 Project Performance

MySQL, SQLite and ClickHouse use the same SQL to create the result in following way:

```
select UNITID, OPEID, CITY, STABBR, ZIP from student
```

SparkSQL yield same result in following way:

```
left.select("UNITID", "OPEID", "CITY", "STABBR", "ZIP")
```

ScalaTion create same result in following way:

```
result = relation.π("UNITID", "OPEID", "CITY", "STABBR", "ZIP")
```

Comparison result is shown in Figure 4.1. ScalaTion and SparkSQL perform the best among these 5. And the on-disk column-oriented database, ClickHouse ranks the second. There is no doubt that column-oriented databases will perform better in project operation. Because the table is stored in columns, there is no need to go through all the rows.

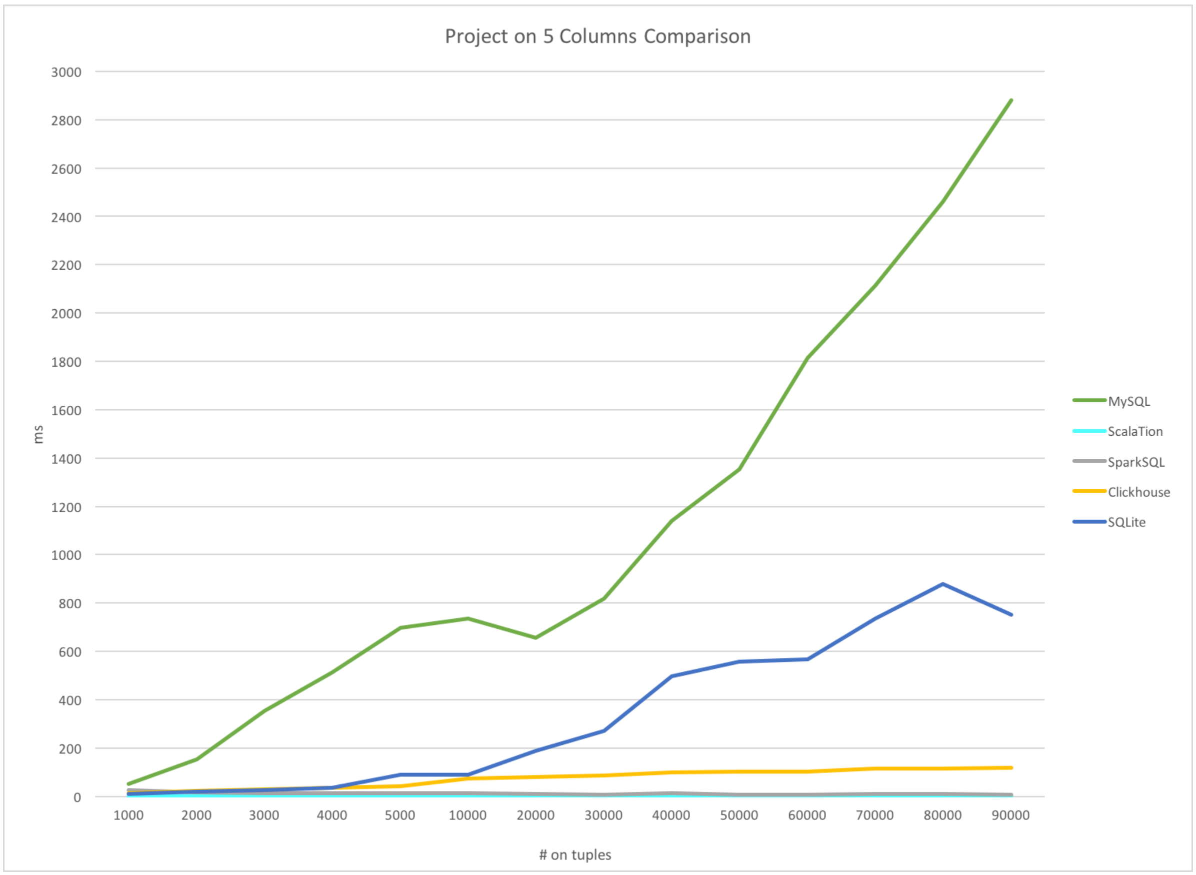

Figure 4.1: Project Performance Comparison

An experiments shows how the different numbers of project columns will affect the performance of project operation. The result is in Figure 4.2. As the number of project column increases, the execution time of ScalaTion is barely affected and the execution time of ClickHouse , SparkSQL and SQLite increase. ClickHouse always performs better than SQLite.

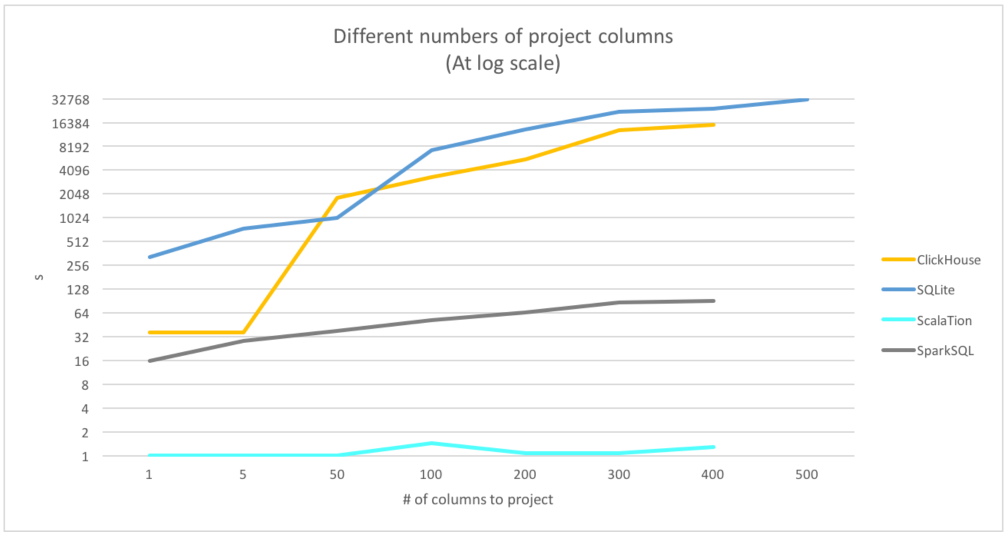

Figure 4.2: The affect of different number of project columns

4.5 Select Performance

MySQL, SQLite and ClickHouse use same SQL to create the result in following way:

```
select * from student where OPEID = 100200;
```

SparkSQL yield same result in following way:

```
student.where(student("OPEID")===100200)
```

ScalaTion create same result in following way:

```
relation.σ("OPEID", _ == 100200)
```

Comparison result is shown in Figure 4.3. ScalaTion and SparkSQL perform the best among these 5. And the in-memory row-oriented database, SQLite ranks the second. ClickHouse performs worse than the row-oriented database SQLite bacause there are 500 columns involved in the result. When there are many columns involved in the query result and few operations are operated on columns, there is no performance gains for column-oriented databases. Because reconstruction of a table is costly in column-oriented databases.

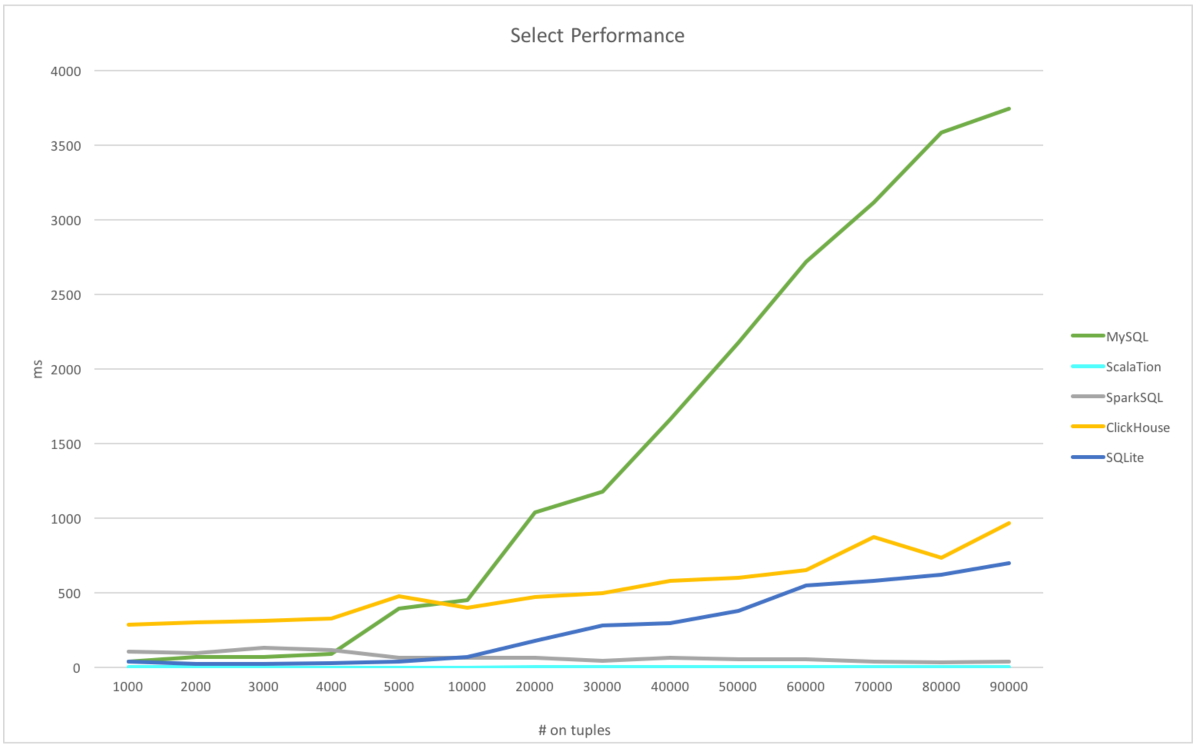

Figure 4.3: Select Performance Comparison

Although column-oriented databases are good at select operation, reconstruction of a table with many columns still make this type of SQL execution costly. ScalaTion are not affected by this factor because ScalaTion has indexes to hash the rows.

An experiment with different number of project columns and different number of selectivity is conducted to show the effect of the number of the project columns to select operation. The number of project columns are 1, 5, 10, 300, respectively. The number of select predicates is 1 and 7, respectively. As Figure 4.4 and Figure 4.5 show, when the number of select predicate is 1 and more and more columns are projected, ScalaTion and SQLite execution time remains stable and execution time of ClickHouse increases. When number of project columns is small and no matter how many predicates are required, ClickHouse will always performs better than SQLite. This comparison shows reconstructing of more and more columns will slow down ClickHouse.

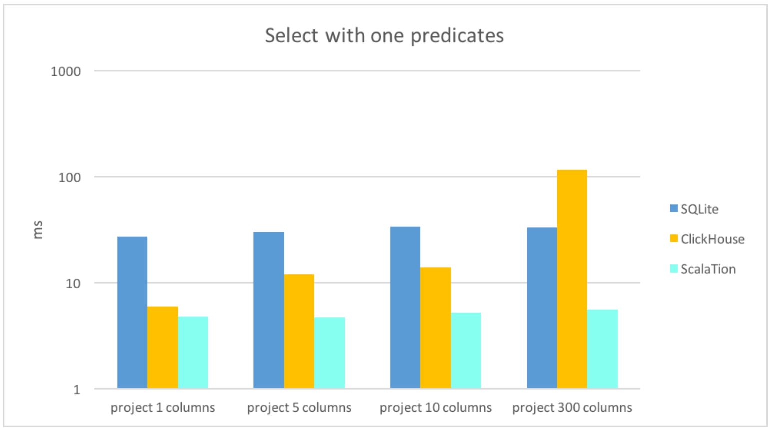

Figure 4.4: Projection with one selection Performance Comparison

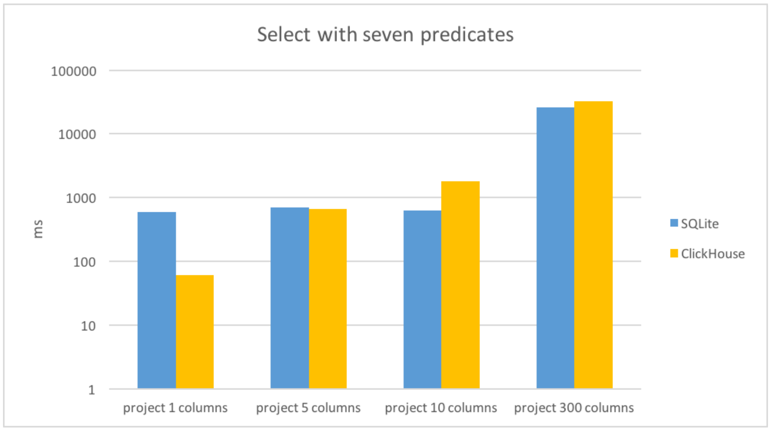

Figure 4.5: Projection with seven selection Performance Comparison

4.6 Union Performance

Next we are going to compare the Union operation. In terms of testing data, we want to union two tables with the same schema.

MySQL, SQLite and ClickHouse use same SQL to create the result in following way:

```
select * from studentA UNION ALL select * from studentB
```

ScalaTion and SparkSQL yield same result in following way:

```
studentA.union(studentB)
```

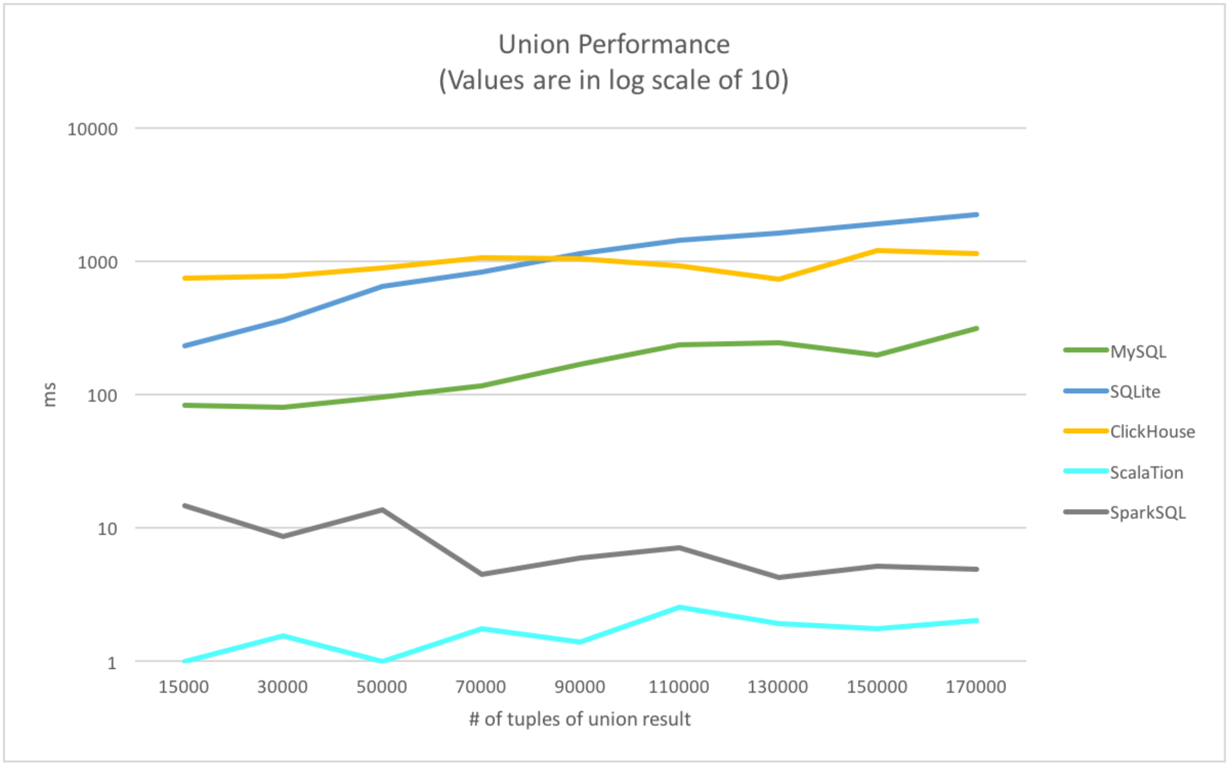

Figure 4.6: Union Performance Comparison

Comparison result is shown in Figure 4.6. Result shows ScalaTion performs 2x faster than SparkSQL. MySQL ranks the third and it uses 100X time as ScalaTion. ScalaTion performs 1000x faster than ClickHouse and SQLite. ClickHouse performs worst among these 5. Again result shows that there is no performance gains in ClickHouse if the SQL involves many columns without many operations on columns.

4.7 Intersect Performance

Next we are going to compare the Intersect operation. In terms of testing data, we want to intersect two tables with same structures.

There is no INTERSECT operator in MySQL and ClickHouse. Similar query can be finished using In clause or EXISTS clause. MySQL and ClickHouse use same SQL to create the result in following way:

```
select * from studentA as A
          WHERE A.Unnamed IN (select Unnamed from studentA);
```

SQLite supports INTERSECT algebra. Same result will be created by following codes:

```
select * from studentA INTERSECT select * from studentB;
```

ScalaTion and SparkSQL yield same result in following way:

```
studentA.intersect(studentB);
```

Comparison result is shown in Figure 4.7.

ScalaTion and ClickHouse, as expected, do not perform as good as row-oriented databases in intersect operation. Since intersect operation needs to go through every value in every row to check if there are any differences. In row-oriented databases, row is stored as a tuple so it is easy to compare. SparkSQL performs the best. The DataFrame in SparkSQL is a Dataset of Row which might have its own compare method, making the intersect operation fast. It is almost 100X faster than average performance of other databases. MySQL and SQLite performs almost the same in this operation.

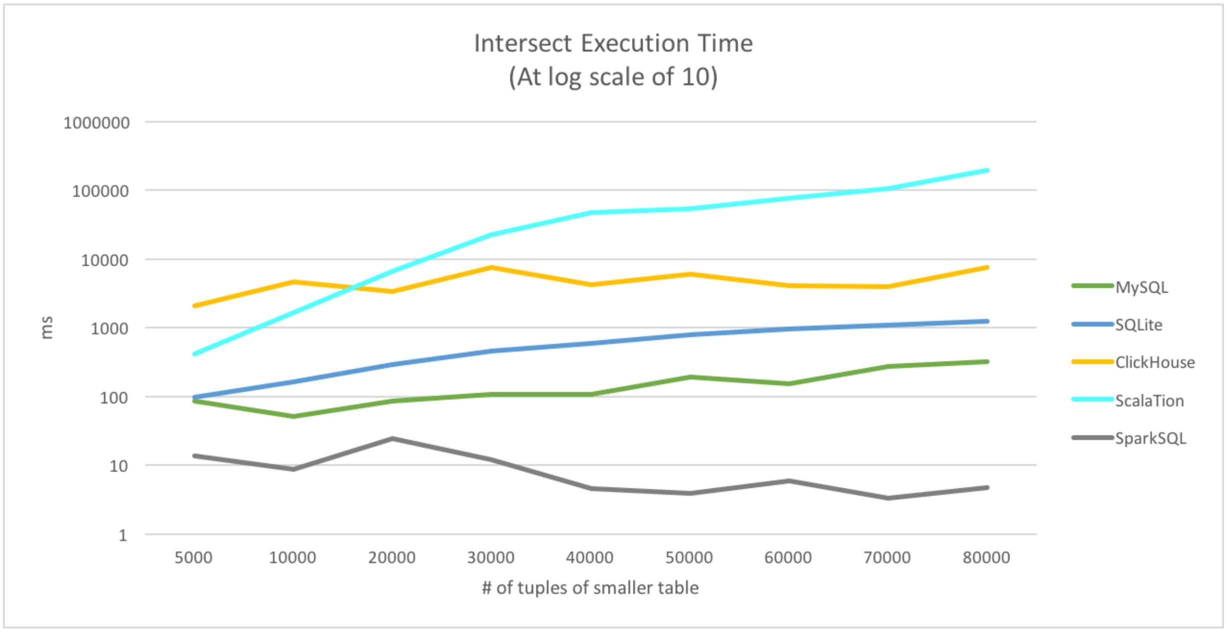

Figure 4.7: Intersect Performance Comparison

4.8 Join Performance

Join operation is not provided in some NoSQL database. Here we compare Join operations.

We use the following codes to get the full information of a student and his detail information.

Same result will be created using MySQL and SQLite by following code:

```
select * from student join address on student.Sid = address.Sid
```

Same result will be created using ClickHouse by following code:

```
select * from student ANY INNER JOIN address USING Sid;
```

Same result will be created using SparkSQL by following code:

```
student.join(address, "Sid")
```

Same result will be created using ScalaTion by following code:

```
val fulltable = student.indexjoin(address, "Sid", "Sid")
```

Comparison result is shown in Figure 4.8. SQLite performs the slowest. SparkSQL performs the best. It shows SparkSQL is good at performing either row-oriented operations such as join or column-orient operations such as select, project. ScalaTion performs second. ScalaTion uses 10X of execution time used by SparkSQL. SQLite surprisingly performs worse than column-oriented databases. Looking into the query plans of SQLite and MySQL, they both use only one index in each query. This to some extend explains the reason why they are slower in index join operation. In ScalaTion, if the join attributes are both primary keys in two tables, two indexes will be used in the index join which explains the reason why ScalaTion is fast in the index join operation. SparkSQL implements the join with hash join using OpenHashMap to leverage the operation. As the size of dataset increases, SQLite performs worse than ClickHouse.

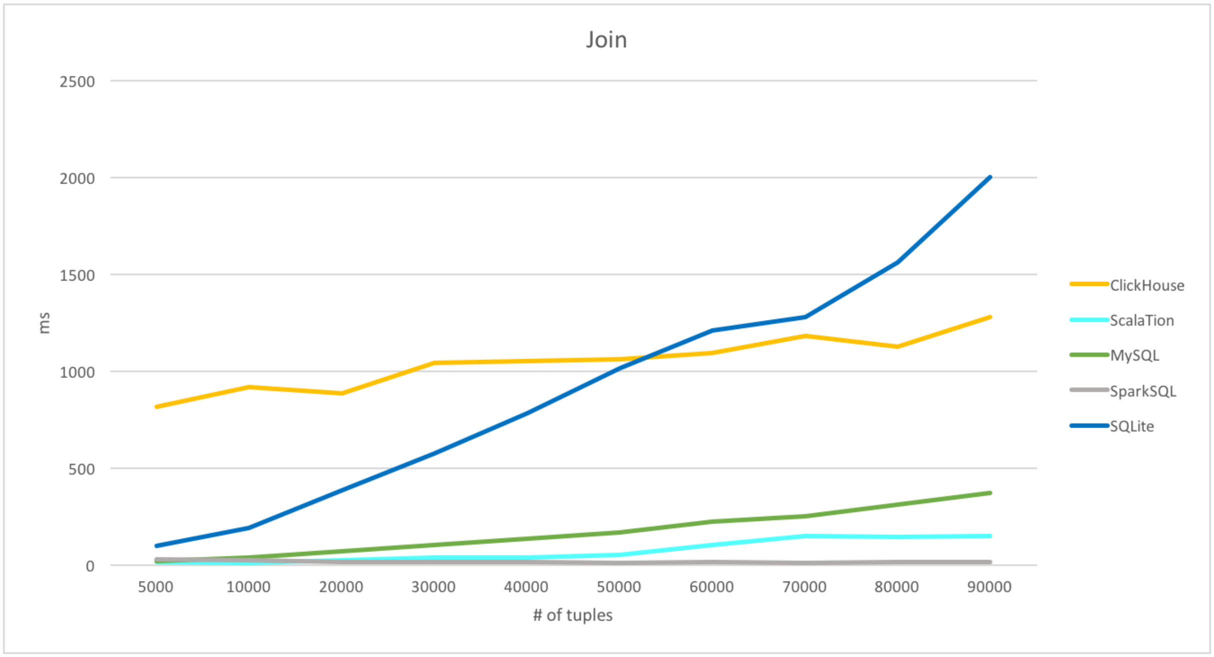

Figure 4.8: Join Performance Comparison

4.9 Parallel Join Performance

Parallel version of index join is implemented in ScalaTion. Parallel index join separates the table into a specific number of partitions and does the operations in parallel.

A similar way to call parallel index join in ScalaTion is shown as following code.

```
val fulltable = student.parjoin(address, "Sid", "Sid", 4)
```

Here, the 4 indicates the table will run in 4 threads. User can define specific thread numbers based on their computer hardware.

For dataset whose size is over 90000 rows, the execution time of the parallel index join operation can reduce to half of the time of index join.

Comparison result of index join and parallel index join is shown in Figure 4.9. As the results shows, when the size of dataset reaches 90000, parallel index join takes only half time of index join.

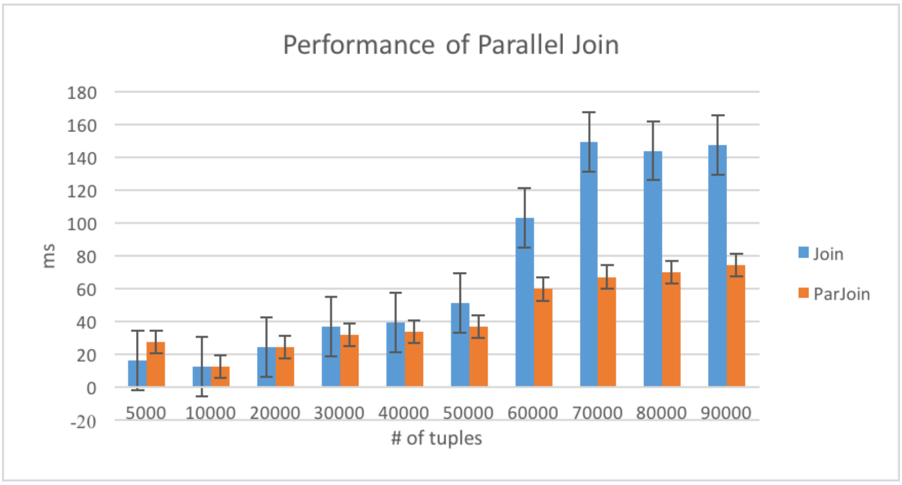

Figure 4.9: Parallel Join Performance

4.10 Aggregation without Grouping Performance

First we test the aggregation function without any grouping. In regards of the testing data, we want to know the total sum of salary of all the students. We compare five ways to get the sum in ScalaTion, MySQL, SQLite, ClickHouse and SparkSQL.

Same result will be created using MySQL, SQLite and ClickHouse by following code:

```
Select sum(salary) from student
```

Same result will be created using SparkSQL by following code:

```
student.agg (sum (student ("salary") ) )
```

Same result will be created using ScalaTion by following code:

```
student.sum("salary")
```

Comparison result is shown in Figure 4.10. Result shows two column-oriented databases are the fastest among the five. ScalaTion and ClickHouse performs the best as expected for aggregation. They are 50X faster than in-memory RDBMS, SQLite and 100x faster than on-disk RDBMS MySQL. SparkSQL ranks the third which uses twice time of ScalaTion.

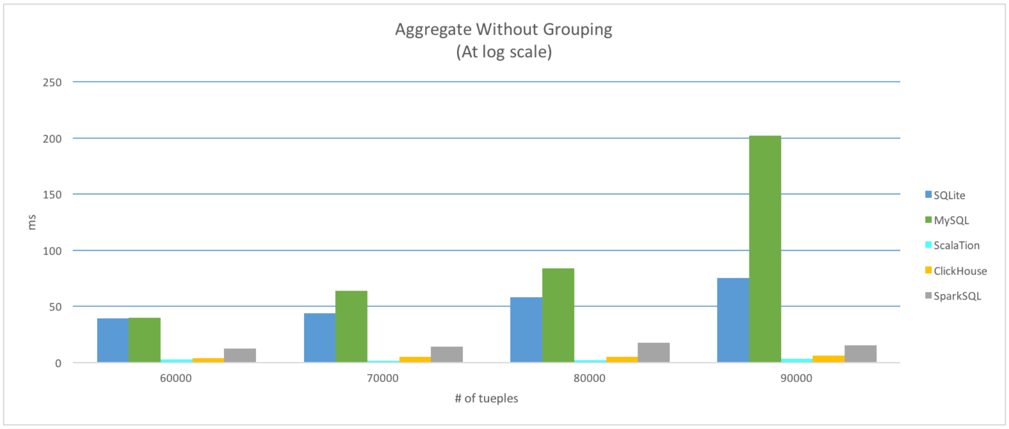

Figure 4.10: Aggregate without Grouping

4.11 Project on One Groupby Column, Aggregation with Grouping Performance

Later, we simply want to group the table by one attribute, aggregate on one attribute and project only on the aggregate column. In regards of the testing data, we want to look at the max of ZIP for each city.

Same result will be created using MySQL by following code:

```
Select max(ZIP) from student group by CITY
```

Same result will be created using SQLite and ClickHouse by following code:

```
Select max(ZIP) from student group by CITY Order by CITY
```

Same result will be created using SparkSQL by following code:

```
val result = student.groupBy("CITY").max("ZIP")
                                    .alias("counts").orderBy(student("CITY"))
```

Same result will be created using ScalaTion by following code:

```
student.groupby("CITY").epi(Seq(max), Seq("ZIP"))
```

Comparison result is shown in Figure 4.11. As expected, the column-oriented database, ClickHouse has a good performance on aggregation. SparkSQL performs the same as ClickHouse. ClickHouse and SparkSQL is 10x better than ScalaTion. ScalaTion performs better than SQLite. MySQL is the slowest among these 5.

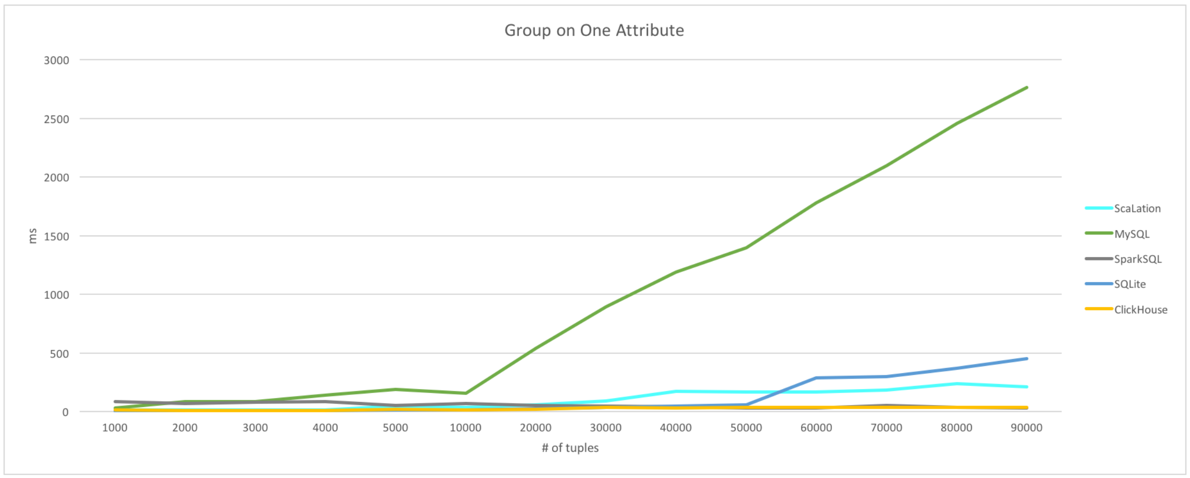

Figure 4.11: Aggregate on One Attribute and Project on Aggregate Column

4.12 Project on Muiltiple Columns, Aggregation with Grouping Performance

Later, we want to test to group the table by one attribute, aggregate on one attribute and project on multiple columns. In regards of the testing data, we want to look at the Sid, CITY and sum of ZIP of each city.

Because Sid is not listed in Group By clause and has no relation with the group by column CITY. MySQL does not support this query when the "ONLY_FULL_GROUP_BY SQL" mode is enabled (which it is by default)[1]. ClickHouse does not support this query either. So we make the query into two different queries and test the performances. The result of Q1 will show a random representative of each group. The result of Q2 will show every tuple in each group.

[1]https://dev.mysql.com/doc/refman/5.7/en/group-by-handling.html

We use following codes to get Sid, CITY, and max of ZIP of student for every CITY using ScalaTion:

```
Q1:
student.groupby("CITY")
            .epiEasay(Seq(max), Seq("ZIP"), "Sid", "CITY")
Q2:
student.groupby("CITY")
            .epi(Seq(max), Seq("ZIP"), "Sid", "CITY")
```

The execution of Q1 in Mysql needs to disable the related SQL mode. Same result will be created using MySQL by following code:

```
Q1:
// After SET sql_mode = ‘’;
Select Sid, CITY, max(ZIP) from student group by CITY
Q2:
select Sid, CITY, max from student
join
(select CITY as c2, max(ZIP) as max from student group by CITY) as B
on CITY = B. c2
order by CITY;
```

Same result will be created using SQLite by following code:

```
Q1:
Select Sid, CITY, max(ZIP) from student group by CITY Order by CITY;
Q2:
select Sid, CITY, max from student
join
(select CITY as c2, max(ZIP) as max from student group by CITY) as B
on CITY = B. c2
order by CITY;
```

ClickHouse does not support types of query such Q1. The columns to project can only be the group by columns or aggregate columns or they must have one-to-one connection to the group by column. Same result can be created by executing Q2 in following code:

```
Q1:
```

```
Not Available in ClickHouse
Q2:
select Sid, CITY, max
from
(select Sid, CITY from student)
ANY INNER join
(select CITY, max(ZIP) as max from student group by CITY) USING CITY
```

SparkSQL does not support types of queries such Q1 either. Similar result will be created using SparkSQL by following code:

```
Q1:
Not Available in SparkSQL
Q2:
val intermediateTable = student.groupBy("CITY").max("ZIP").alias("counts")
val result = student.select("Sid","CITY")
      .join(intermediateTable, "CITY").orderBy(student("CITY"))
```

The set of Q1 performance shows in Figure 4.12. Result shows ScalaTion is 10X faster than MySQL and 3X faster than SQLite in processing types of query such as Q1.

The set of Q2 performance shows in Figure 4.13. ClickHouse shows almost same performance as ScalaTion. ScalaTion is 5X faster than SQLite, 8X faster than SparkSQL and 10X faster than MySQL.

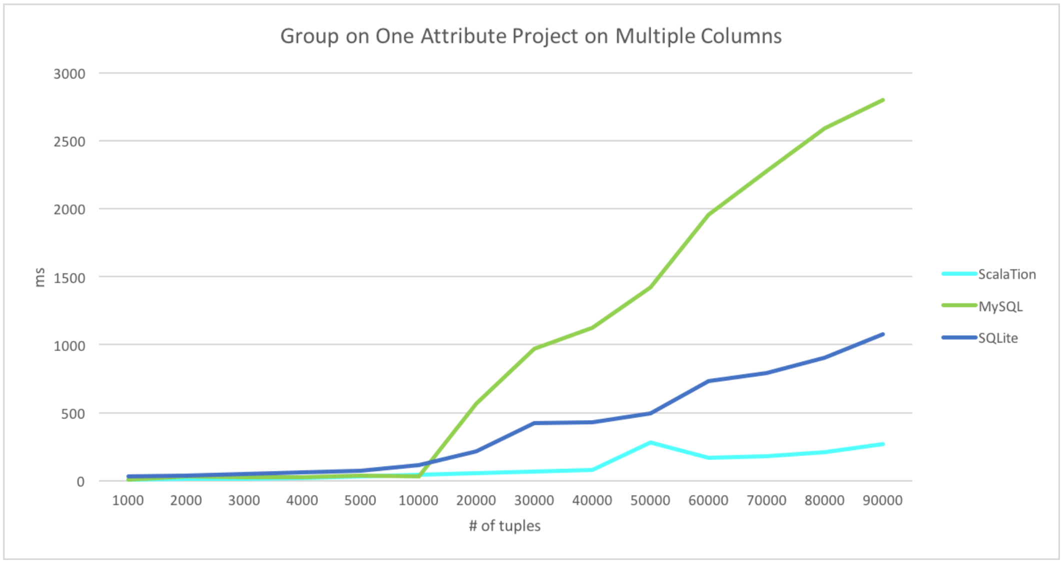

Figure 4.12: Sets of Q1 Performance

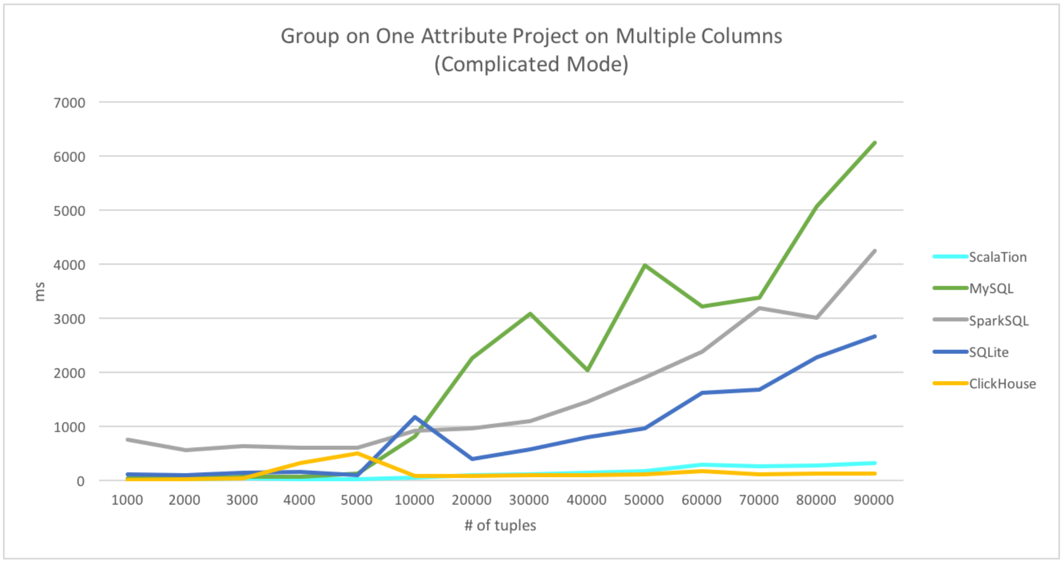

Figure 4.13: Sets of Q2 Performance

4.13 RelationFrame High Order Functions Performance

In following 4 sections, we are showing the performance result of Map, Reduce, Filter of RelationFrame in Relation, DataFrame in R, DataFrame in Scala and DataFrame in Spark. We use a data file with size of 90000 rows.

4.14 Map Performance

Map function takes the dataframe and apply a user define function to the data structure and return the value. In terms of test data, we want to add 5 to every value in the Unnamed column.

Result will be created using following code in R DataFrame:

```
plus <- function(x){return(x+5)}
res <- sapply(student[,c("salary")], plus)
```

Result will be created using following code in Python DataFrame:

```
res = map(lambda x:x+5, student[‘salary’])
```

Result will be created using following code in Spark DataFrame:

```
student.map{ case Row(Unnamed:Int, case_statud:String, position:String,
                       city:String, job_title:String) => (salary + 5) }
```

Result will be created using following codes in ScalaTion RelationFrame:

```
def plus(v1:Int): Int =  v1 + 5
val result = student.map(plus, "salary")
```

Result is in Figure 4.14. As the result shows, Python DataFrame performs the best among these 4. RelationFrame is almost as good as Python DataFrame and only takes 1/10 of time R does. The following is the Spark DataFrame. R DataFrame performes the worst.

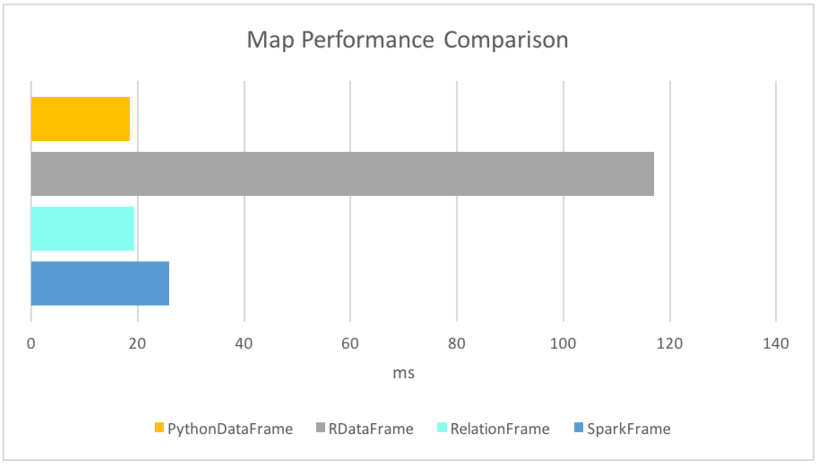

Figure 4.14: Map function performance of size of 90000

4.15 Reduce Performance

Reduce function apply a user define function to the data structure and return a higher level perspective of the column. In terms of test data, we want to add up all the values in Sid column of student.

Result will be created using following codes in R DataFrame:

```
plus <- function(x,y){return(x+y)}
res <- Reduce(plus, student[,c("salary")])
```

Result will be created using following codes in Python DataFrame:

```
res = reduce(lambda x, y: x+y, student['salary'])
```

Result will be created using following codes in Spark DataFrame:

```
student.select("salary").reduce((x,y) =>
                    Row(x.getInt(0) + y.getInt(0)))
```

Result will be created using following codes in ScalaTion RelationFrame:

```
def plus(v1:Int, v2:Int): Int =  v1+v2
val result = student.reduce(plus, "salary")
```

Result is in Figure 4.15. The result shows RelationFrame performs the best and the following is Python DataFrame. R needs almost 10X of execution time as RelationFrame and Python DataFrame does. Spark DataFrame needs almost 20X of execution time as RelationFrame and Python DataFrame does.

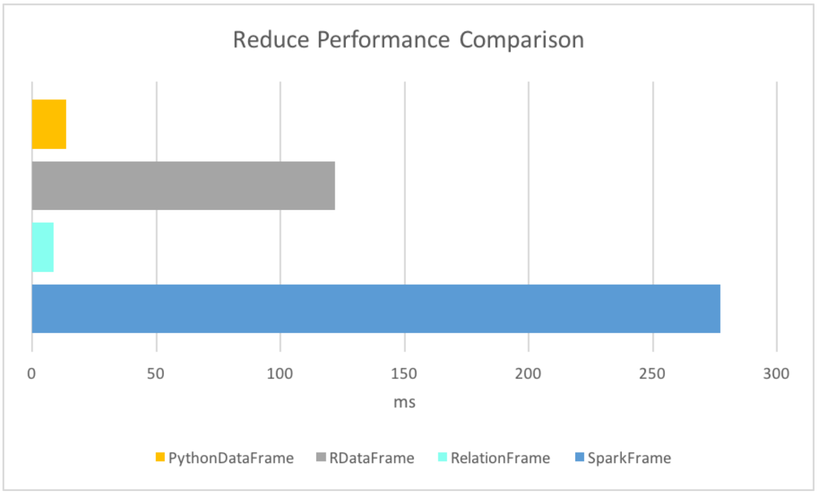

Figure 4.15: Reduce function performance of size of 90000

4.16 Fold Performance

Fold function is similar to Reduce function with a default value. R, SparkSQL and Python do not have separate API for fold method. SparkSQL handles fold method as Reduce function with an initial value.

In terms of test data, we want to add up all the values in Sid column of student with a default sum 10.

Result will be created using following codes in R DataFrame:

```
plus <- function(x,y){return(x+y)}
res <- Reduce(plus, student[, c("salary")]) + 10
```

Result will be created using following codes in Python DataFrame:

```
re = reduce(lambda x,y: x+y, student, 10)
```

Result will be created using following codes in Spark DataFrame:

```
student.select("salary").reduce((x,y) =>
     Row(x.getInt(0) + y.getInt(0))) + 10
```

Result will be created using following codes in ScalaTion RelationFrame:

```
def plus(v1:Int, v2:Int): Int =  v1+v2
val result = student.fold(10, plus, "salary")
```

Result is shown in Figure 4.16.

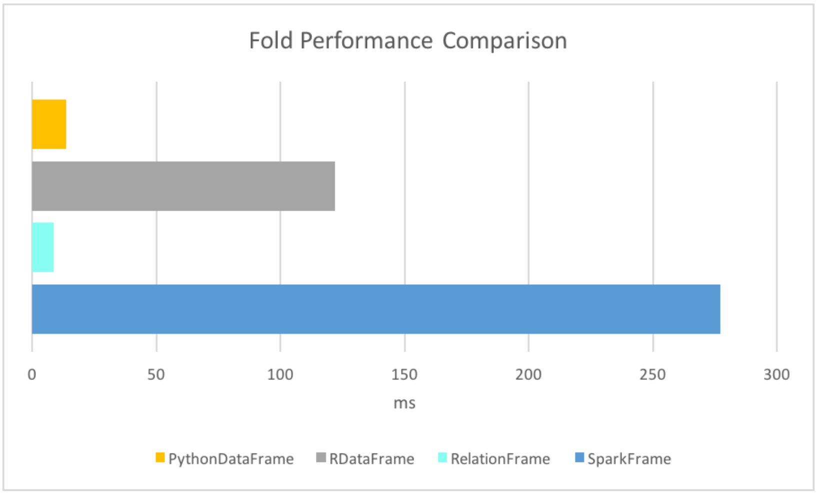

Figure 4.16: Fold function performance of size of 90000

4.17 Filter Performance

Filter function goes through the whole data structure and return a new data structure based on specific filtering predicate. In terms of test data, we want to get a new student table whose OPEID attribute has all values equal to 100200.

Result will be created using following codes in R DataFrame:

```
res<-student[student$OPEID == 100200,]
```

Result will be created using following codes in Python DataFrame:

```
res = student['OPEID'] == 100200
```

Result will be created using following codes in Spark DataFrame:

```
val res = student.filter(student("OPEID").=== 100200)
```

Result will be created using following codes in ScalaTion RelationFrame:

```
student.filter( _ == 100200, "OPEID" )
```

Result is in Figure 4.17. RelationFrame performs the best among these 4. Python DataFrame ranks second. Spark DataFrame and R DataFrame takes 10X to 20X time as RelationFrame does.

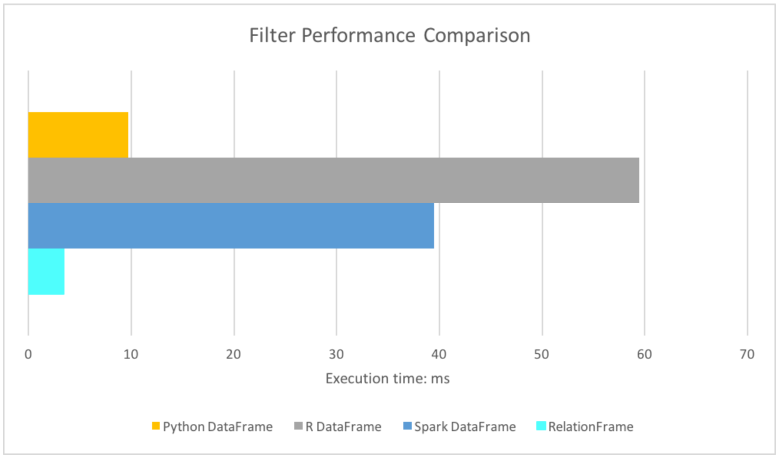

Figure 4.17: Filter function performance of size of 90000

4.18 Pipeline Performance

We have another set of comparison which are consisted of some common processes of a data processing before data modeling. The process includes filtering on the raw data based on a specific range and then join two different feature table by a common key. We compare this pipeline process time of ScalaTion and MySQL using the dataset described in Section4.2.

Result of the comparison is shown in Figure 4.18. ScalaTion uses half time of MySQL for the total process.

Do regression using ScalaTion:

```
val x = fulltable.toMatriD(0 to 29)
val y = fulltable.toVectorD(30)
val rg = new Regression(x, y, Cholesky)
// use QR Factorization
```

Do regression using R:

```
student <- dbReadTable(conn = con, name = 'student')
```

fit <- lm(mn_earn_wne_p6 ∼ CONTROL+PREDDEG +UGDS + INEXPFTE + PPTUG_EF +C150_4+ AVGFACSAL + ADM_RATE_ALL + SAT_AVG + TUITIONFEE_IN + TUITIONFEE_OUT+ UGDS_BLACKNH + UGDS_API + UGDS_AIANOld + UGDS_HISPOld + INC_PCT_LO + INC_PCT_M1 + INC_PCT_M2 +INC_PCT_H1 + INC_PCT_H2 + PAR_ED_PCT_1STGEN + PAR_ED_PCT_MS + PAR_ED_PCT_HS + PAR_ED_PCT_PS + pct_grad_prof + female + first_gen + age_entry + DEBT_MDN, data=student)

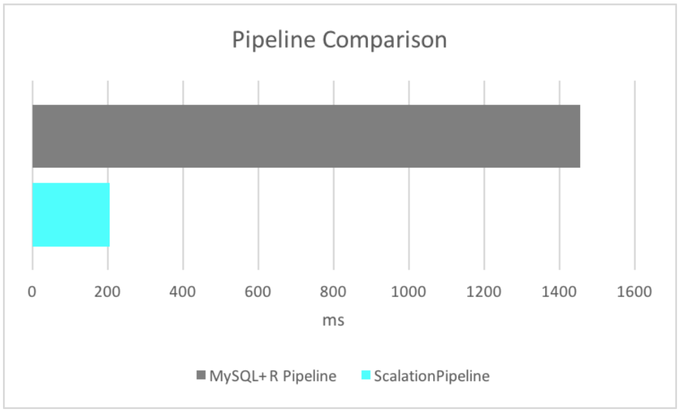

Figure 4.18: Preprocessing Pipeline Compare

Chapter 5

RelatedWork

Industry has been searching for relational processing and procedural processing engines for Big Data for years. One of the engines which can perform these functions in a stable way is Shark [23]. Shark runs on Spark and offers both relational query processing and advanced analytics. Later SparkSQL [24] has developed on Shark with a more programmer-friendly API and with a richer library. It also comes with an optimizer, Catalyst. Catalyst is one of the first production quality query optimizer built on Scala. It uses many features of Scala such as pattern-matching [25] and quasiquotes [26]. It also helps SparkSQL to optimize logical plans, generate physical plans, generate code, and compare plans based on costs.

Other open-source libraries which have similar functions and extensions to handle UDFs are Hive [27] and Pig [28]. Although these tools are designed to work on clusters, they share the same tasks with ScalaTion which are to provide fast analytics on Big Data in a SQL-like way. Analytics on big data focuses on the performance of three common operations: groupby, aggreagte and join. The finely designed column-oriented storage, particular design of query processing component, and optimization strategies of these systems share benefits with ScalaTion. Hive's strategy of partial aggregation on skewed data and hash-based partial aggregation in Hive has helped improve its performance of aggregation. Pig has a "safe" optimizer which will apply database style optimization in most cases. However, when performance benefits are uncertain, operations will be executed as the order written by the user.

The process of reconstructing the columns back into a tuple is called materialization in column-oriented databases. Strategies for materialization are critical to column-oriented databases. Different strategies about when to rebuild the columns back to tuple and their tradeoff are discussed in detail in paper [13]. Vertica has implemented plenty of different materialization strategies and paper [29] has a thorough discussion about them. These ideas help column-oriented systems save a significant amount of time and space during queries processing by reducing intermediate results.

"Database cracking" technique [30, 31] is a technique which is based on the idea that maintaining indexes as query is being processed. Rather than revising or reconstructing the original table, system can create a copy of a segment of table which the query touches and manipulate on the copy. Later the system will combine results of all the copies and build a new table from it. C-Store [14] adopted this technique in their implementation. It inspired the optimization of groupby and aggregation operations in ScalaTion.

Chapter 6

Conclusions and Future Work

Column-oriented databases store data as columns. This enables the database to apply compression on columns and to retrieve all the values from a column at once without scanning the entire table. The performance of project, union and select operations of ScalaTion are several hundred times faster than open-source in-memory row-oriented databases. Especially the select operation performs better than ClickHouse and SparkSQL. The aggregation operation is competitive with ClickHouse and is 4X faster than SQLite. After building indexes, the index join operation which is not provided in some NoSQL databases performs 2X faster than row-oriented databases and 1000X faster than the column-oriented database, ClickHouse.

The aggregation operation marks the table into different groups and then calculates the result based on each group [32]. Before the groupby process, the columns are stored in the order of the insertion (basically randomly). So in knowing how to efficiently find the aggregated values based on the group which they belong to, becomes a main task. Some column-oriented databases store different projections of tables in different sorting orders [14], making this process easier. In ScalaTion, aggregation functions are optimized to be 4X faster than the in-memory row-oriented database SQLite.

The ScalaTion column-oriented database provides an easy and quick way to load data into the database as a Relation. The Relation API provides a rich library of relational operations to handle analytics. And it can be easily transformd into RelatonFrame which has high order functions to support most of the common operations in a data pipeline. The API supports user-defined functions to manipulate data at higher level in a customized way. Based on the performance evaluation, the filter operation is tremendously fast compared with open-source RDBMS. Also Relation is also closely combined with modeling in ScalaTion, providing a way for users to finish data processing within ScalaTion. According to the comparison result of two pipelines, using a ScalaTion analytic pipeline and ScalaTion modeling is faster than using traditional RDBMS combined with other machine learning packages such as R [20] or Scikit [33].

When datasets reach a certain size, partitioning the table into different sections and applying parallelism can help the system improve performance. Parallelism is currently provided for join, select operation. As of the current version, ScalaTion does not support update and delete operations due to the clumsy nature of writing to column-oriented database [6]. These can be implemented in the future. Select with multiple predicates can be leveraged by using bitmap as Druid did. Intersect operation is now implemented in quadratic algorithm and later can be optimized using hashing. The matrix is now arranged as row-oriented in ScalaTion. Transforming the relation to matrix will be significantly faster if the matrix is organized as column-oriented. A further study needs to be made upon the benefits and tradeoff between using column-oriented or row-oriented matrix in various matrix operations.

Query optimzer is a future development direction yet to be realized. A stable and complex optimizer can help the system reduce unneccessary intermediate results based on cost or rules. SparkSQL has extensible query optimizer Catalyst [24]. Now query optimization in ScalaTion still needs to be done manually. Techniques for leveraging query processing typically using column-oriented indexes and techniques which designed specifically for column-oriented databases about speeding up relational operations are disscussed in paper [34]. Another optimization direction is the usage of vertorization. ClickHouse has combined vectorization [35] in their system to leverage their query processing. JAVA-based systems are still out of reach of these skills since JVM has not supported vectorization yet. Project Panama [36] under OpenJDK is trying to accommodate the support of vectorization into JVM.

The ScalaTion column-oriented database and RelationFrame is open source at `http://www.cs.uga.edu/~jam/ScalaTion.html`.

Bibliography

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[2] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, October 1981.

[3] James Ong, Dennis Fogg, and Michael Stonebraker. Implementation of data abstraction in the relational database system ingres. *SIGMOD Rec.*, 14(1):1–14, September 1983.

[4] Oracle timeline. `http://www.oracle.com/us/corporate/profit/p27anniv-timeline-151918.pdf`, 2007.

[5] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. pages 249–264, 1974.

[6] V. N. Gudivada, D. Rao, and V. V. Raghavan. Nosql systems for big data management. In *2014 IEEE World Congress on Services*, pages 190–197, June 2014.

[7] A. B. M. Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *CoRR*, abs/1307.0191, 2013.

[8] Ann M. Kelly Daniel G. McCreary. *Making Sense of NoSQL A guide for managers and the rest of us.* 2013.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[11] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[12] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proc. VLDB Endow.*, 2(2):1664–1665, August 2009.

[13] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475, April 2007.

[14] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[15] Overview of clickhouse architecture. `https://clickhouse.yandex/docs/en/introduction/distinctive_features/`.

[16] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.

[17] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012.

[18] Martin Odersky, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

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

[20] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[21] Python. `https://www.python.org/`.

[22] Qiang Hao. Feature selection and classification of post-graduation income of college students in united states.

[23] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[24] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[25] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 273–298, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] Martin Odersky Denys Shabalin, Eugene Burmako. Quasiquotes for scala, a technical report. Technical Report MSU-CSE-06-2, 2013.

[27] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[28] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[29] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1196–1207, April 2013.

[30] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *In CIDR*, 2007.

[31] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 4(9):586–597, June 2011.

[32] John Miles Smith and Diane C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst.*, 2(2):105–133, June 1977.

[33] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent

Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[34] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 59–72, New York, NY, USA, 2009. ACM.

[35] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM.

[36] Project panama. `http://openjdk.java.net/projects/panama/`.

Appendix

Relation API User Manual

1. Load CSV into Relation

```
val testRelation = Relation(path, "testRelation", domain, keyindex)
```

2. Project of Relation

```
result = relation.pi("AttributeA", "AttributeB",
                                       "AttributeC", "AttributeD")
```

3. Select of Relation

```
relation.("AttributeA", _ == value)
```

4. Union of Relation

```
tableA.union(tableB)
```

5. Intersect of Relation

```
tableA.intersect(tableB)
```

6. Join of Relation

```
val result = tableA.indexjoin(tableB,
                           "AcolumnAttribute", "BcolumnAttribute")
```

7. Aggregate of Relation

```
table.groupby("AttributeA")
student.epi(Seq(max), Seq("AggregateAttribute"),
                                    Seq("AttributeA", "AttributeB"))
```

8. Make Relation into MatriD

```
val  x = tableA.toMatriD(1 to 1000)
```

9. Make Relation into VectorD

```
val y = tableA.toVectorD("tableAAttribute")
```

Appendix

RelationFrame API User Manual

1.Crearte a RelationFrame from a Relation

```
val relationFrame = RelationF(testRelation)
```

2. Map of RelationFrame

```
def udf(v1:Int, v2:Int): Int =  v1+v2
val result = student.map( udf, "AttributeA", valuev2)
```

3. Reduce of RelationFrame

```
def udf(v1:Int, v2:Int): Int =  v1+v2
val result = student.reduce( udf, "AttributeA" )
```

4. Fold of RelationFrame

```
def udf(v1:Int, v2:Int): Int =  v1+v2
val result = student.fold( defaultvalue, udf, "AttributeA" )
```

5. Filter of RelationFrame

```
relation.filter( _ == value, "AttributeA")
```

6. User-defined function of RelationFrame

```
relation.map( (v1:Int, v2:Int) =>  v1+v2, "AttributeA")
```