Relational Computing Using HPC Resources: Services and Optimizations

Manikandan Soundarapandian

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Madhav V. Marathe, Chair
Sandeep Gupta
Keith R. Bisset
Jiangzhuo Chen
Anil Vullikanti

July 27, 2015
Blacksburg, Virginia

Keywords: distributed database, HPC, supercomputers, postgres-xc, computational epidemiology

Relational Computing Using HPC Resources: Services and Optimizations

Manikandan Soundarapandian

(ABSTRACT)

Computational epidemiology involves processing, analysing and managing large volumes of data. Such massive datasets cannot be handled efficiently by using traditional standalone database management systems, owing to their limitation in the degree of computational efficiency and bandwidth to scale to large volumes of data. In this thesis, we address management and processing of large volumes of data for modeling, simulation and analysis in epidemiological studies. Traditionally, compute intensive tasks are processed using high performance computing resources and supercomputers whereas data intensive tasks are delegated to standalone databases and some custom programs. DiceX framework is a one-stop solution for distributed database management and processing and its main mission is to leverage and utilize supercomputing resources for data intensive computing, in particular relational data processing.

While standalone databases are always on and a user can submit queries at any time for required results, supercomputing resources must be acquired and are available for a limited time period. These resources are relinquished either upon completion of execution or at the expiration of the allocated time period. This kind of reservation based usage style poses critical challenges, including building and launching a distributed data engine onto the supercomputer, saving the engine and resuming from the saved image, devising efficient optimization upgrades to the data engine and enabling other applications to seamlessly access the engine . These challenges and requirements cause us to align our approach more closely with cloud computing paradigms of Infrastructure as a Service(IaaS) and Platform as a Service(PaaS). In this thesis, we propose cloud computing like workflows, but using supercomputing resources to manage and process relational data intensive tasks. We propose and implement several services including database freeze & migrate and resume, ad-hoc resource addition and table redistribution. These services assist in carrying out the workflows defined.

We also propose an optimization upgrade to the query planning module of postgres-XC, the core relational data processing engine of the DiceX framework. With a knowledge of domain semantics, we have devised a more robust data distribution strategy that would enable to push down most time consuming SQL operations forcefully to the postgres-XC data nodes, bypassing its query planner's default shippability criteria without compromising correctness. Forcing query push down reduces the query processing time by a factor of almost 40%-60% for certain complex spatio-temporal queries on our epidemiology datasets.

As part of this work, a generic broker service has also been implemented, which acts as an interface to the DiceX framework by exposing RESTful APIs, which applications can make use of to query and retrieve results irrespective of the programming language or environment.

Dedication

எல்லாப் புகழும் இறைவனுக்கே!

A.R. Rahman (Musician)
My biggest inspiration

Translation from Tamil:
All the praises and honours that I've received and will receive are not mine and they belong only to the Almighty!

Acknowledgements

I would have never been able to finish my thesis without the guidance of my committee members and support from my friends and family.

Foremost, I would like to express my sincere gratitude to my advisor, Dr. Madhav V. Marathe for his continuous support during my Master's study and research. He was always ready to help and provided valuable suggestions in all my decision-makings throughout the past two years at Virginia Tech.

I offer my sincere gratitude to my mentor, Dr. Sandeep Gupta, who has supported me throughout the course of the thesis work with utmost patience. I will cherish my experience working with him for the rest of my career, and I hope to continue to collaborate with him in the future.

I would also like to thank my other committee members: Dr. Keith Bisset, Dr. Jiangzhuo Chen and Dr. Anil Vullikanti for offering valuable comments about my thesis.

I am grateful for my mother and father for their continuous encouragement and support for whatever decisions that I make in life.

This work has been partially supported by NSF NetSE Grant CNS-1011769, DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, and DTRA CNIMS Contract HDTRA1-11-D-0016-0004.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Driven by technological advancement and globalization, today's world is increasingly interconnected and mobile. Long distance travel across cities, states, and, countries in large numbers are common across almost all parts of the globe. This, however, has also created a critical threat where an outbreak of an infectious disease in a locality can rapidly engulf large sections of population covering vast geographical regions. If the disease is life threatening, as was the case with the recent Ebola outbreak in West African countries, it can cause significant loss of life, adversely undermine population health, and stress public infrastructure. Clearly, timely and effective response, when faced with such threat is paramount.

Computational epidemiology is an innovative scientific discipline which aims to address this challenge. It predicts how communicable diseases spread, and evaluates, in real time, the responses designed to curtail the spread. Marathe et al. [42] state that the spread can be understood as spatio-temporal diffusion through a population and that the goal of computational epidemiology is to develop computer models that capture this diffusion. Over the past several years, computational epidemiology has developed the theory and practice of agent-based models and their simulations which accurately capture this spatio-temporal diffusion in a given population.

Such agent-based models are large and complex. Moreover, their simulations produce outputs that are massive in size. This is partcularly true when modeling a large population spread across a vast geographical area. The temporal aspect of the problem also play a significant role in scaling the problem size by several folds. Hernan et al. [33] state that the size of epidemiological data is becoming large resulting in unique and never-to-be replicated resources.

[37] states the size of epidemiology datasets including a synthetic population, contact network

Data	Size	Storage
Synthetic Population	566 (GB)	Relational
Social Network and Simulation Output	1.84 (TB)	File

Table 1.1: Size of epidemiology data

and agent based simulation output of influenza for the USA and UK as given in table 1.1. The simulation output is typically generated as flat files containing the disease exposed time, infected duration and recovery time following the SEIR model for every individual in the synthetic population. Such astronomical data sizes pose challenges in storage and management as well as in fast and efficient query processing.

Computational epidemiology, thus involves a variety of tasks ranging from being highly compute intensive to highly data intensive. A combination of both is also not uncommon. Generating disease models with a variety of parameters and running simulations on synthetic populations is a largely computation intensive task given how complicated the generated disease model is and the sizes of the synthetic population and social contact network. On the other hand, analysis on the simulated infection data can be deemed highly data intensive which is either done using SQL queries or through other data mining and querying techniques. Moreover, these analyses are typically done through a map or graph based web front-end which requires real time response rates. Currently, these data intensive epidemiological analyses are handled by loading data onto a traditional standalone database and launching a series of complex SQL queries with time consuming joins and aggregation operators. Compute-intensive computational epidemiological tasks are handled using supercomputers and other High Performance Computing (HPC) systems as they are guaranteed to provide very high-level computational capacity because of their sophisticated hardware resources.

1.2 Motivation

Varshney et al. [51] claim that data-driven decision making has much promise for effectively responding to large scale disease outbreaks. Google Flu Trends [10] is an attempt to model flu spread and trends over the past year using the rate of Google searches regarding flu. The results of the data driven analysis are shown in the form of spatio-temporal maps and plots. Several government surveillance agencies including the Center for Disease Control (CDC) try to take efficient measures to track and predict such outbreaks by performing repeated simulations with different intervention strategies, disease model parameters and stochasticity. When these situation assessments and spatio-temporal simulations are performed over a global scale, it leads to producing hundreds of terabytes of data. As per the Worldometers [9] population reading, at the time of this writing, global population is 7.3 billion and steadily rising. So, when these situation assessments and simulations are done with synthetic data

for such a huge population, it produces even more hundreds of terabytes of simulation and infection data. Performing analysis on such large datasets is a real challenging task.

Li et al. [38] claim that traditional standalone databases cannot effectively handle large datasets and scaling out data is a viable solution to introduce parallelism in processing. They proposed a PC-cluster based solution to partition data so as to support query parallelism. Such distributed data processing has effectively helped in faster query processing.

As stated before, web front ends used to analyse epidemiological assessments must be highly responsive and with the massive amounts of data to query, scalable high speed distributed database frameworks will be required or else these systems will be rendered useless. FluCaster [2] is a pervasive web application developed at **NDSSL, VBI** for disease forecasting and situation assessment using agent-based stochastic simulation. It displays predictions of influenza rates on a map and graph based web front-end and is currently deployed on a standalone Oracle database. FluCaster issues many spatio-temporal queries, a class of queries that processes data by both time and space to display the predictions. Because of the inadequacy in processing large volumes of data and execute queries with real time response rates on the standalone Oracle database, several summary tables were created aggregating the infection counts at state, county and block group level. These summary tables were created after analysing the kind of queries that the application issues. Though these additional tables help in speeding up the queries, they add to the already existing large volume of synthetic population and simulation data. Such problems are endemic with standalone databases, but can be solved by scaling out data using a distributed database and utilizing parallelism in query processing.

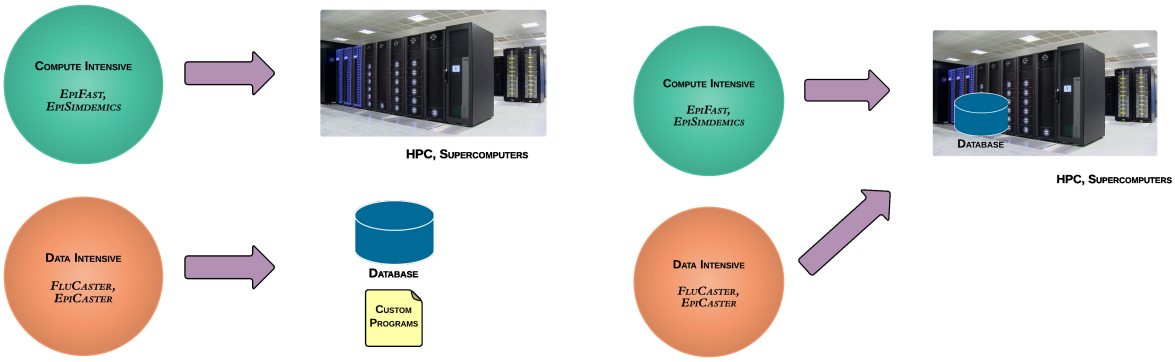
1.3 Contribution

In this work we address some of the challenges faced when carrying out a large scale epidemiological study. In particular, we address management and processing of data for analysis in large scale epidemiological studies. Currently, this management and processing is handled using a variety of resources, primarily, supercomputers, high performance clustered file system called General Parallel File Systems (GPFS) [50], and standalone databases. Even though model generation and simulations are performed in parallel within the supercomputers with all its inputs and outputs maintained either in GPFS file system or as tables in standalone relational databases, still all the data processing and analysis is handled either via custom written hard wired programs or via a database engine. When it is done using a database engine, the data is stored in the form of relational tables in the database and the processing is expressed as a series of SQL statements. We term such workloads, expressed as a sequence of SQL statements, as "relational" workloads. Following are some examples of relational workloads in computational epidemiology analysis.

- In order to perform analysis on the simulated data that is stored in databases, relational

workloads are imminent. Say, we need to find the total infection count for a particular state during a particular month in the future; it can be queried from the database using SQL statements. Such spatio-temporal queries are integral to any computational epidemiology analysis.

- When several predictions of a certain disease infection needs to be shown on a plot to display the trends produced by each prediction model, the data has to be queried from the relational database that is storing the infection data using appropriate relational workload. Some common examples for relational workloads in spatio-temporal analysis of diseases deployed by the application FluCaster[2] are given in Appendix B.



(a) Traditional Approach (b) Our approach

Figure 1.1: Handling compute intensive and data intensive tasks - Traditional approach vs Our approach

The focus of this thesis is to address management and processing of data for efficiently running these relational workloads present in computational epidemiology. The main hypothesis is to utilize supercomputing resources to scale processing of these relational workloads. DiceX [37] is an attempt to follow this hypothesis and make use of supercomputing resources for relational computing. Generally compute intensive tasks are processed using high performance computing resources whereas data intensive computing is still handled either by custom written hard-wired programs or by using standalone databases. With the exception of exploratory graph analysis, almost all computations can be expressed in relational terms and processed using databases. However, standalone databases are limited by resources and are quite expensive for handling large volumes of data. Figure 1.1 contrasts our approach with the traditional methodology.

Also, a standalone database is "always" on, i.e., a user can submit queries at any time and get the required results. With supercomputers, resources are not always immediately available and must be acquired, again, only for a fixed time period. The resources similarly must be released either when the computing is completed or at the expiration of the allocation time

period. This reservation based usage style poses challenges and therefore it is not trivial to launch a data processing engine onto a supercomputer. Some major challenges posed are as follows:

- Build a distributed data processing engine
- Request supercomputing resources and launch the data engine
- Freeze and migrate the engine as an image to persistent storage upon resource expiration time
- Resume the engine from preserved image
- Optimize the engine for high speed query execution
- Enable other applications and webapps to connect to the engine and execute its relational workload on the engine

These challenges and requirements make our approach relatively closer to the 'cloud computing' paradigms of Infrastructure as a Service(IaaS) and Platform as a Service(PaaS). Henceforth, we propose 'cloud computing' like workflows but using supercomputing resources to manage and process relational data intensive tasks. These workflows broadly aid in supporting the following use-cases:

- Acquire supercomputing resources, launch relational data, run queries and save results to GPFS
- Acquire supercomputing resources, launch relational data, save data in persistent storage
- Resume experiment using the already saved data from the persistent storage upon acquiring resources from the supercomputer
- Acquire supercomputing resources, launch relational data or resume from saved data, conduct experiments using relational workloads, seek and extend supercomputing resources in an ad-hoc fashion

This thesis proposes several services that assist in carrying out the above mentioned worklflows. DiceX freeze & migrate and resume utility service deploys an efficient high speed algorithm based on MPI I/O to quickly migrate the distributed database from the supercomputer to a remote persistent storage and also to resume the database engine from the preserved image when the experiment needs to be resumed. DiceX node add utility aids in cluster elasticity and flexible resource management in supercomputers by means of acquiring new resources from the supercomputing ecosystem in an ad-hoc manner in order to efficiently

scale out data for improved throughput on executing the complex relational workloads. We implement these services and perform scalability and performance analysis of them which is explained in detail in chapters 4 and 5.

Postgres-XC, the core relational data engine of the DiceX framework performs query push down to maximize parallel execution of queries. But, not all the SQL operations of a complex query can be pushed down to the individual postgres-XC data nodes, therefore not helping much in reducing the execution time of complex relational workloads. We propose and implement an optimization upgrade to the query push down criteria of postgres-XC based on a knowledge of domain semantics, utilizing this knowledge to efficiently distribute data so as to support complete push down of queries to the data nodes. We conducted experiments with some common computational epidemiology relational workloads to test the performance of the proposed new query push down criterion and it has been found that it supports much faster query execution times compared to the traditional push down criteria enforced by postgres-XC.

Finally, we implement a generic interfacing broker service that exposes RESTful APIs for applications and webapps to make use of the DiceX framework irrespective of their programming language or environment. The broker service is aimed at functioning as a resource manager that keeps track of all the currently available distributed data engines running in the supercomputer and also to route the queries from the web front-ends or other applications to the corresponding data engine of interest. The applications, irrespective of the programming language or environment, can efficiently make use of this broker service for fast querying and retrieval of epidemic data.

1.4 Organization of thesis

Challenge	Chapters
Freeze and migrate the engine	chapter 4
Resume from saved image	chapter 4
Re-size the engine	chapter 5
Optimize the engine	chapter 6
Connect to the engine from applications and webapps	chapters 9 & 10

Table 1.2: Thesis Organization

This thesis reviews some related studies pertaining to scaling out analysis in computational epidemiology and popular distributed data processing frameworks in chapter 2. In chapter 3, the general architecture and services provided by the DiceX framework has been elucidated. The remainder of the thesis addresses the challenges in utilizing supercomputing resources for relational computing that we have mentioned before and this organization is shown in table 1.2.

Chapters 7 and 8 study similar distributed systems and exploit their merits and demerits with a deeper analysis, and finally chapter 11 provides concluding remarks and lists possible future extensions.

Chapter 2

Related Work

2.1 Overview

In the following sections, we will review other works pertaining to scaling out and distributed data processing in data intensive sciences including computational epidemiology. We will also explore the state of the art computational approaches and associated tools and technologies.

2.2 Scaling out computational epidemiological tasks

Eubank [28] used the empirical estimates of social network and contact patterns produced by TRANSIMS, a large scale simulation of transportation systems, to design and implement a novel system to simulate the spread of diseases among individuals in a large urban population. This simulation system was built only for some major cities in the USA using their transportation network, and was scaled out to be run in a PC-cluster of 16 CPUs due to the amount of computation involved. Episimdemics [15] is a scalable parallel algorithm to simulate the spread of contagion in large, realistic social contact networks using individual-based models. Such individual-based simulations are highly compute intensive and therefore need to be scaled out if it is to be performed using a large synthetic population. Episimdemics deploys a MPI based distributed memory parallel implementation on mid-size HPC systems to scale this compute intensive task. Epifast [17] is another scalable parallel algorithm based on a novel interpretation of the stochastic disease propagation in a contact network and is implemented using a master-slave computation model allowing scalability on distributed memory systems. Bisset et al. [19] implemented CharmSimdemics, that simulated the spread of infectious diseases over large scale realistic social networks using Charm++ [1], a machine independent parallel programming system that provides high-level mechanism and strategies to facilitate the task of developing highly complex parallel applications. CharmSimdemics

was tested on a 768 core system and achieved up to a 4-fold increase in performance compared to the MPI based Episimdemics.

2.3 Computational epidemiology frameworks

Now let us review some frameworks explicitly built for performing large scale computational epidemiology experiments. Bisset et al. [18] proposed Indemics, an interactive data intensive framework and modeling environment for pandemic planning, situation assessment, and course of action analysis. Indemics supports online interactions between users and high performance simulation engines and was extended from Epifast. It integrates database management systems with simulation engines using a database to store information that is required to drive simulations as well as to support complex interventions and queries. Corley et al. [23] implemented a computational framework of multiple sources and computational modules to evaluate what-if scenarios and quantify public health actions. They validated the underlying model of their framework by taking into account the demographic and behavioral differences in the population along with the disease dynamics, morbidity, and mortality. While the previous two frameworks were focused largely on the computation aspect, Shamimul et al. [32] implemented a data mapping framework for computational epidemiology datasets in a digital library which can aid in data management and warehousing. This framework enables the flexibility to switch between various databases and execute queries on them.

2.4 Big data distributed computing frameworks

Big data frameworks like MapReduce and Hadoop have been extensively utilized in any field that involves processing of large datasets. So, with our previous claims describing the large volumes of data involved in computational epidemiology, it is an obvious choice to explore and review these big data distributed computing frameworks. Dean et al. [25] state MapReduce is a programming model for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Every large computation can be expressed in terms of a *map* and *reduce* function and the underlying runtime system automatically parallelizes the computation across large scale commodity clusters. The MapReduce framework also handles machine failures and schedules inter-machine communication to make efficient use of the network and disks. Doulkeridis et al. [27] presented a survey of various large scale analytical query processing using the MapReduce framework and reviews the state of the art in improving the performance of parallel query processing. Yang et al. [53] focused on the inability or inefficiency of the MapReduce framework when applied to relational operations like joins. They attempted to add a new merge phase that can efficiently merge data already partitioned and sorted by map and reduce modules. [13] discusses on strategies to optimize

joins in a MapReduce environment specifically on analytic queries in which a very large fact table is joined with smaller dimension tables.

Hadoop [3] is the open source implementation of the MapReduce framework developed at Apache foundation. It allows for distributed processing of large data sets across clusters of computers using simple programming models. It uses the Hadoop Distributed File System (HDFS) which provides high throughput access to the application data. Apache Hive [4] is a data warehouse built on top of Hadoop which facilitates querying and managing large datasets residing in the distributed storage. It provides a mechanism to project structure onto data and query using a SQL like language called HiveQL. It also helps to efficiently plug in custom mappers and reducers when it is inefficient to express the logic in HiveQL. Fegaras et al. [29] state that when such custom mappers and reducers are used in place of queries in HiveQL, it might result to suboptimal, error-prone, and hard-to-maintain code. They present an optimization framework for SQL like MapReduce queries and introduce an alternative query language called MRQL which is expressive enough to capture most of the computations in declarative form and at the same time is amenable to optimization. Dryad [35] is a general purpose distributed execution engine for coarse grain data parallel applications developed at Microsoft Research. It is designed to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of cores. DryadLINQ [54] is a system and a set of language extensions that enable a new programming model for large scale distributed computing. It generalizes common execution environments like SQL, MapReduce, and Dryad by adopting an expressive data model of strongly typed .NET objects and supporting general purpose imperative and declarative operations on datasets with a traditional high level programming language.

Apparently most MapReduce based data warehousing and processing tools provide very limited options for query optimizations. Unlike databases, these tools do not support the concept of schema or indexes, which are important to speed up the process of accessing large volumes of data. Even though Hive supports indexing, it is implemented as tables and the query must be rewritten using the newly created index tables, which becomes too expensive in case of really large tables. Using a parallel or distributed database like postgres-XC in place of MapReduce frameworks, will tend to be much faster because of the various query optimization techniques that are inherent to databases.

2.5 Other Domain specific approaches

Having reviewed MapReduce and other distributed frameworks, in this section, we review some novel distributed computing systems specially designed for and catering to specific domains other than computational epidemiology.

Transportation

Chen et al. [22] developed a MapReduce based distributed modeling framework for use as a traffic forecasting system. This framework proposed data-driven traffic flow forecasting to meet the computation and data storage requirements. Huang et al. [34] designed and implemented a parallel map matching algorithm for GPS data on a Hadoop platform and presented the performance of their approach on a GPS dataset of more than 120 billion GPS records.

Bioinformatics

[36] presents a comprehensive review on the usage of distributed computing frameworks in the field of bioinformatics. Li [39] implemented a distributed and parallel implementation of a bioinformatics tool ClustalW which helps in aligning multiple protein or nucleotide sequences. Owing to the need for faster analysis on large datasets, Li implemented the tool ClustalW-MPI using the MPI library and made it run on distributed workstation clusters as well as traditional parallel computers.

Machine learning and artificial intelligence

Gillick et al. [31] used Hadoop on modest sized compute clusters to evaluate its efficiency in running machine learning tasks. The authors have also listed the classes of machine learning problems that can be solved using the MapReduce or Hadoop framework. They claim that Hadoop can be a good choice for basic machine learning operations on large datasets but not for complex learning tasks. Low et al. [40] extended the GraphLab framework which is an asynchronous, dynamic. graph parallel computation on a shared memory setting to use a distributed setting without compromising data consistency. [21] showcases methods and approaches in using distributed computing on robotics.

Multipurpose novel systems

Beynon et al. [16] proposed a distributed data processing framework called DataCutter which is designed to support for subsetting and processing of datasets in a distributed and heterogeneous environment. DataCutter can be utilized for data-intensive applications on a diverse number of fields including sensor networks, medical imaging, etc.

2.6 Distributed and parallel databases

In this section, we review some state of the art parallel and distributed databases. Parallel database systems can be described as a database management systems implemented on a multiprocessor system with a high degree of connectivity. A distributed database is a collection of multiple logically interrelated databases distributed over a network. Li et al. [38] proposed InfiniteDB which aims at storing and processing of massive databases and to support high performance analysis. InfiniteDB supports parallelisms of intra-query, inter-query, intra-operation, inter-operation, and pipelining along with support for parallel data warehousing and data mining. It employs an efficient data de-clustering mechanism to partition a relation into subsets and distribute them among the nodes in a cluster. ScaleBase [8] is another proprietary framework with a core relational database cluster built on MySQL/InnoDB storage and optimized for cloud and web scale applications. It supports ACID compliance, two phase commit and rollbacks, and an efficient SQL query model including cross database joins and aggregations. SCOPE [56] is another distributed computation system combining benefits from both traditional parallel databases and MapReduce execution engines to allow easy programmability, massive scalability and high performance through advanced optimization. SCOPE deployed a separate optimizer that converts SQL scripts into efficient execution plans for distributed processing. Bubba [20] is a highly parallel computer system for data intensive applications. It is based on a scalable shared nothing architecture and can scale up to thousands of nodes. Postgres-XC, which is the integral relational data processing engine of the DiceX framework [37] is a multi-master write-scalable postgreSQL cluster on a shared nothing architecture. It supports more than 3x scalability performance speedup with just five servers, compared with pure postgreSQL (DBT-1) [6]. [26] compares the performance of traditional databases with Cloud Database i.e. database in the cloud environment. The comparison was made between the traditional database **Microsoft SQL server 2008 R2** and the cloud database deployed on **Windows Azure**. It has been found that data acquisition, data management, integrity of data storage, data mining, resource allocation and management, and database migration are some of the major factors that could affect the performance of cloud databases. Apparently, traditional databases performed better than cloud databases for larger datasets used in the research.

2.6.1 Data distribution strategy

In distributed database systems, data is usually physically distributed among the database nodes of the cluster by fragmenting or replicating it. Fragmentation can be done in various ways including random projection or sharding based on a specific rule. Data placement or layout of the database in a distributed framework is deemed an important resource management issue in shared nothing parallel database systems. Mehta et al. [43] explored data placement issues in detail and demonstrated that advancement in hardware technology allows declustering of data without penalizing performance. Ping [47] states that a good data

placement strategy could improve execution efficiency of multi-join queries. He also claims that the bandwidth of network communication is always the bottleneck of parallel database system based on PC clusters and data communication among nodes would create more time cost when executing join operations.

2.6.2 Snapshot and resume in distributed databases

Even though save and restore from backups is a common phenomenon in traditional standalone databases, it is not an easy task with respect to distributed databases because of the amount of data involved and varied data placement strategies. We review some of the previous researches in snapshotting and resuming service in distributed databases. [44] presents an instant loading methodology that allows scalable bulk loading from CSV files in main memory databases at wire speed. Adiba et al. [12] discusses various snapshot semantics and implementations and shows the importance of the snapshotting concept in centralized and distributed databases. [46] showcases a new checkpointing scheme for distributed database systems with the concept of 'shadows' of data items to make sure that the collected data item values are transaction consistent.

2.7 Elastic cluster

Now let us review some of the researches in elastic resource management in cloud computing, as we are looking at implementing the same idea of cluster elasticity in DiceX promoting ad-hoc, dynamic addition of new data nodes without disrupting its services. Park et al. [45] presented a novel elastic resource provisioning to expand the capacity of a cluster by dynamically creating virtual machines from cloud pools. [24] introduces ElasTras, an elastic, scalable, and self-managing transactional database for the cloud environment. It scales to thousands of application databases or tenants, effectively consolidating tenants with small footprints while scaling out large tenants across multiple servers in a cluster. [30] presents a comprehensive survey of elasticity mechanisms across both commercial and academic solutions in the field of cloud computing.

Chapter 3

DiceX Framework

DiceX framework is a one-stop solution for distributed database management and processing. It allows for easy deployment of computing engines on a supercomputer cluster, and the deployed engine enables parallel processing of relational expressions to achieve throughput and scalability. Previous research [37] has shown that this approach performs significantly better than standalone relational databases such as Oracle and other distributed frameworks like Hive on complex spatio-temporal queries involving time consuming SQL operations like `joins` and `group bys` over relatively larger datasets. As part of this thesis work, DiceX has been improved both from the standpoint of optimization as well as the stability and usability of the framework by addressing all the key challenges in utilizing supercomputers for relational data intensive computing. In this chapter, we will review the DiceX architecture and its prominent features, and describe the possible workflows enabled by the framework.

3.1 Architecture

DiceX has a simple and elegant design and can be easily configured to handle high performance data intensive operations, in particular, relational data intensive computation. Figure 3.1 depicts the architecture of DiceX framework. The core engine used in DiceX is Postgres-XC (or PGXC) which is a master-slave style distributed data engine. Postgres-XC is a multi-master, write scalable, shared nothing postgreSQL cluster. DiceX uses a toned down version of the data processing engine with only one master and several slave compute nodes.

The architecture of postgres-XC is key to understanding its scalability capabilities and its limitations. It architectural components are:

- Global Transaction Monitor(GTM)
  It handles the transactions from all coordinators and data nodes. A GTM-proxy can

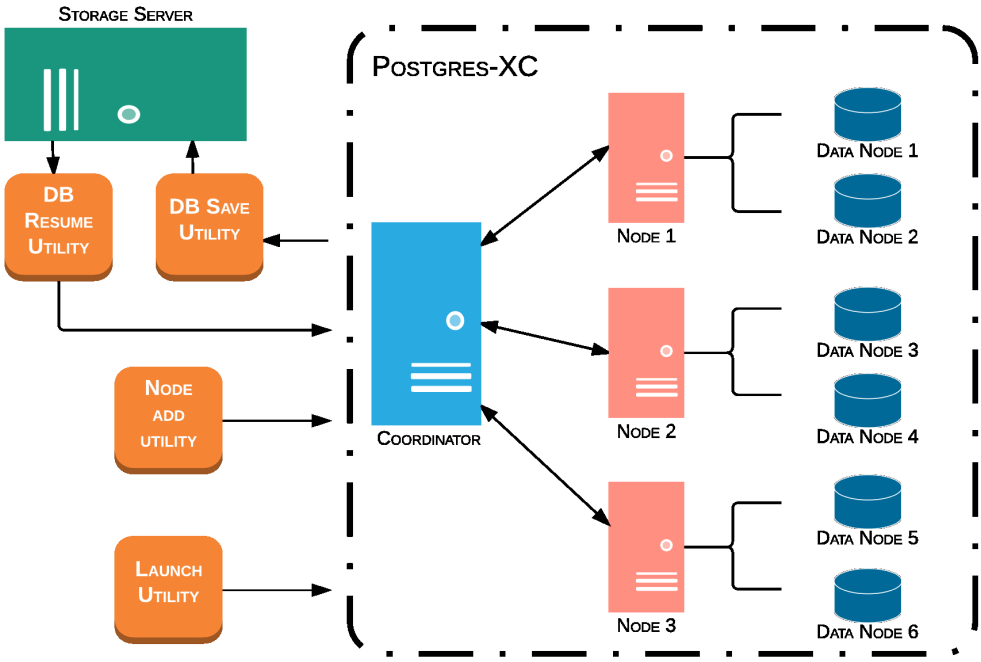


Figure 3.1: DiceX Framework Architecture

be configured to reduce the number of interactions and amount of data sent to GTM. It issues global transaction identifiers to transactions to maintain a consistent view of the database on all nodes and provides ACID properties.

- Coordinator
  The coordinator runs in a separate server, manages the user sessions and acts as an interface that helps to interact with the GTM and data nodes. The coordinator takes care of parsing the user query and generating the query plan before pushing down the query to the data nodes. It does not store any data.

- Data Node
  The data nodes store the actual data and there are several data distribution strategies which can be decided based upon requirements. In order to ensure high availability, standbys can be configured as a fault tolerant mechanism.

As shown in figure 3.1, DiceX is a collection of services that enables relational data intensive computing within a supercomputer. These services include abilities to configure and launch distributed data engine, ingest remote data from relational databases, and efficiently query spatio-temporal data. New features including freeze & migrate and resume from a relational database image, and ad-hoc addition of new resources have been added as complimenting features to DiceX as part of this thesis work.

3.2 Cluster and cloud based relational computing

Current distributed database solutions involve either using a custom built cluster or cloud services from popular vendors. The cluster based solutions like *Teradata*, *Greenplum*, *Paraccel* and *Vertica* typically offer an 'always on' dedicated database management system. These solutions provide a direct in-place replacement for standalone databases and often carry a fixed cost. On the other hand cloud based solutions are not always on and are available on a reservation basis. The user has to log in and request resources. The cost of using such a system is normally a function of the usage and therefore is variable. These cloud based solutions offer data processing in the following two ways:

- Virtual Machine (VM) based
  In this approach, the cloud service provides a VM built with the operating system, database and web packages of choice. This approach provides faster access to data but involves a long wait time to get the VM booted up every time user logs in. [41] compares the startup delay incurred by popular cloud service providers like Amazon, Windows Azure and Rackspace for varied VM image sizes. It has been found that the fastest solution provided by Rackspace to boot a VM image of size 4GB was 260 seconds which is quite long from an end-user's perspective.

- Global database
  The global databases like Amazon's *DynamoDB* does not provide the user with their own VM, instead providing access to the database that is typically stored in a distributed manner across the network. Unlike the VM based solutions, they have less database startup time but the major disadvantage is that these solutions are often key-value based and do not provide a full SQL solution.

3.3 Cloud computing like workflows in DiceX

While the cluster and cloud based solutions have been well established, launching a data processing engine on supercomputers is not well studied. DiceX is an attempt to utilize supercomputing resources for relational data computing. Supercomputers involve reservation based usage style like cloud services but with a fixed cost and there is no readily available data processing engine or database management system running on them. DiceX enables cloud computing like data processing within supercomputers and we have identified three workflows suitable for users with different requirements. These workflows are illustrated in figure 3.2. A user with a relational query (or a collection of queries) over very large datasets can utilize the workflow(s) suiting their specific requirements.

- Workflow 1
  The most obvious and common workflow is where a user seeking the services of DiceX

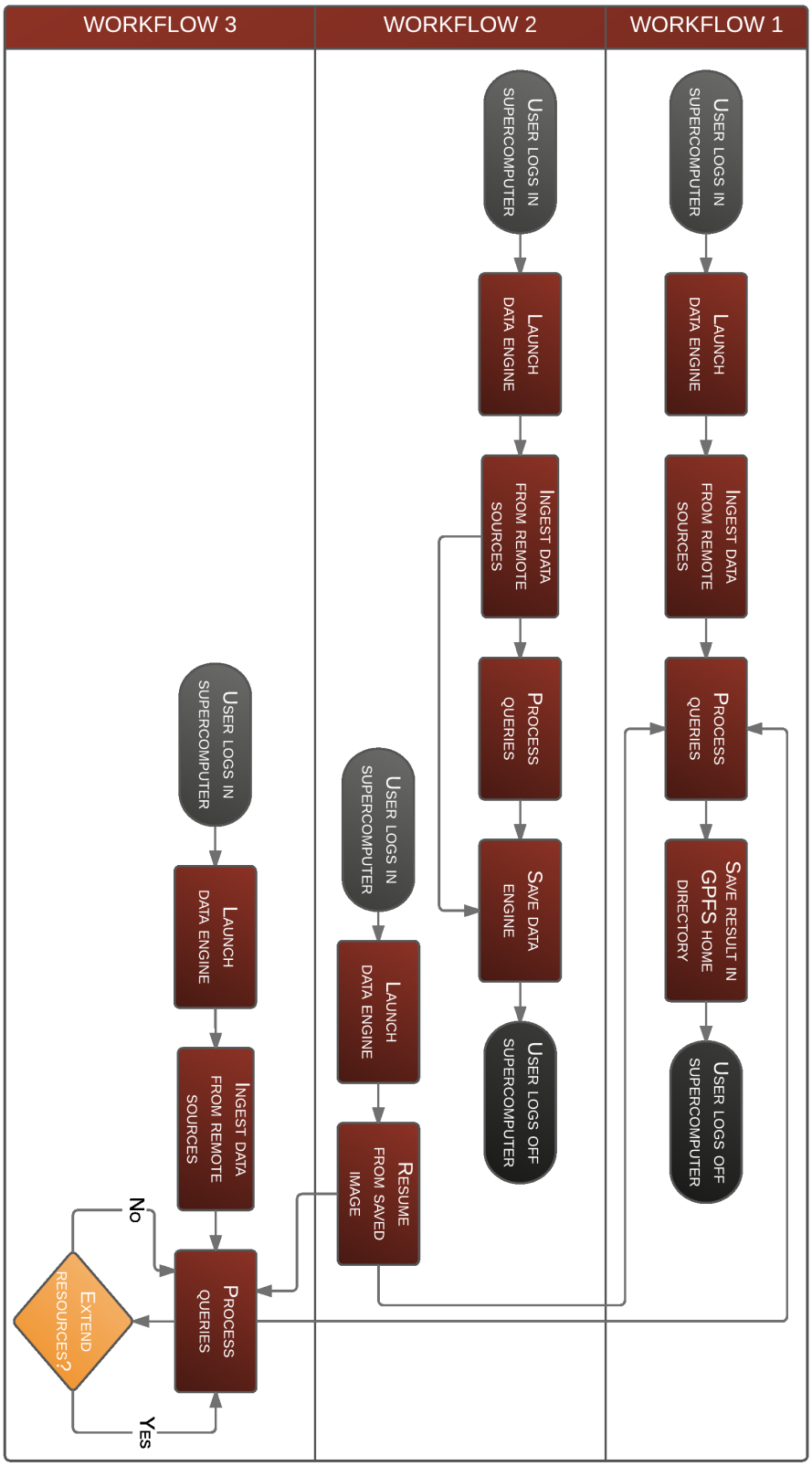


Figure 3.2: DiceX Workflows - Cloud computing like workflows within Supercomputers

logs into the supercomputer, launches his data from remote sources and after executing the queries saves all the results in a local GPFS directory. This workflow is most suited for any user who wants to make use of the scalable distributed relational data engine to perform analysis and get results.

- Workflow 2
  In this workflow the user, after running his experiments, can take an image of the data by making use of the freeze & migrate service provided by DiceX. The same user can log back into the supercomputer and this time, instead of loading data again from remote sources can actually resume his work from the database image that he saved before and continue processing relational queries. Workflow 2 promotes resuming experiments from where it was left off at the end of the last supercomputing session and saves time from loading all data again from the remote source. This workflow, for example, suits users who want to design an effective intervention strategy for controlling the spread of an epidemic. The user can do so by repeatedly running simulations with various intervention strategies where each new iteration in the run is based on the results from the previous iteration. This kind of intervention design can take weeks and even months and this workflow can efficiently aid in continuing the experiment seamlessly. More details on this freeze & migrate and resume service is given in chapter 4.

- Workflow 3
  The user, after launching his data on a set of previously requested supercomputing nodes, can extend the number of nodes in an ad-hoc manner without having to stop any services. After extending the resources and redistributing the relational data, he can continue processing queries. This workflow, for example, can be helpful in the following scenario. Say user A is currently performing analysis on a disease model for the United States. Now A wants to extend it to a global scale by loading the global population data without compromising the running time of the queries. By simply extending the number of data nodes and either ingesting or redistributing data as per requirements, analysis can continue to run for the global population. More details on ad-hoc extension of resources and cluster elasticity is given in chapter 5.

All three workflows are illustrated in figure 3.2. It should be noted that workflows 2 and 3 are new to DiceX and have been implemented as part of this thesis work. These workflows provide a cloud computing like flavor to the DiceX framework.

3.4 Prominent features

Prominent features of the DiceX framework are listed below.

3.4.1 Data distribution

In a distributed data processing environment, data is normally scaled out by partitioning large volumes of data onto several compute nodes on which the database is launched. In such distributed databases, data placement is important as it can effectively help in maximizing the parallelism in execution of any SQL statement and at the same time reduce the communication overhead. Postgres-XC, the core relational data processing engine allows table data to be distributed to the data nodes in a number of ways [7]. They are:

- Replication
  Every row of the table is available in all the data nodes i.e. an exact copy of the table is available in all the data nodes.
- Hash(`col_name`)
  Every row of the table is placed on the hash value of the specified column. Columns of all data types except floating point can be used for distribution by hash.
- Modulo(`col_name`)
  Every row of the table is placed on the modulo value of the specified column. Columns of all data types except floating point can be used for distribution by modulo.
- Round robin
  Every row of the table is placed in one of the data nodes in a round robin fashion. This is somewhat a random distribution of rows of the table.

When none of the above distribution modes is specified, a column with UNIQUE constraint will be chosen by default as the distribution key. If no such column is found, then table rows are distributed in round robin fashion.

3.4.2 Data ingestion

DiceX supports multi-view parallel data ingestion from a relational database as well as copying from flat files using SQL's `COPY` command. The multi-view approach has been implemented in order to reduce the loading time of large datasets by just a series of `insert` statements. From experimentations, it has been found that both the multi-view parallel data ingestion and `COPY` command take almost the same amount of time to ingest data, with the `COPY` command being an order of magnitude faster than the multi-view ingestion. These are the only two data ingestion facilities available in DiceX to load data and the freeze & migrate and resume feature introduced in chapter 4 is a complimentary feature for data loading.

3.4.3 Indexing and clustering

DiceX employs efficient indexing and clustering methodologies with a motive to improve query processing time. B-tree indexing is enforced on the tables as it has been found that it is best suited in improving query execution time on spatio-temporal data, and in particular, computational epidemiology data. Clustering of table data is also employed to reduce the disk accesses by placing all related rows together on the same disk space [37].

3.4.4 Data intensive computing on DiceX

DiceX can efficiently aid in forecasting, visualization and analysis of spatio-temporal data common to computational epidemiology. Previously, spatio-temporal queries with complex aggregates and filters which do not scale well enough on standalone databases have been executed using the DiceX framework and they performed much better [37]. As part of this thesis work, the core processing engine of the framework has been optimized further based on a knowledge of domain semantics and better data distribution strategies. This will be discussed in chapter 6.

Chapter 4

DiceX - Freeze & Migrate and Resume

When the postgres-XC distributed database engine is stopped at the end of a current supercomputing session, all associated postgres-XC services including the coordinator and data nodes are stopped and the table data that were created and distributed among the data nodes are destroyed. DiceX framework currently does not provide a feature to freeze and migrate the current snapshot of data, so that it can be reused later when DiceX is launched again. This freeze & migrate and resume utility will be immensely helpful in addressing the key challenge in utilizing supercomputers for relational computing. This utility service is more generic and can be used in many diverse scenarios including performing post simulation analysis on data generated from disease simulation as and when resources are available, storing intermediate parameters involved in a multi step large scale simulation where the simulation has to be performed over a long time period and supercomputing resources need to be relinquished and acquired again. This service can also aid in testing and validating an experimental process where data needs to be freshly loaded every time for running each test scenario. Moreover, this service cannot be deemed as application specific because of the multi-faceted usage that it offers.

4.1 Motivation

The motivation behind implementing this freeze & migrate and resume utility service is to handle the following:

- Reduce database start-up time as currently tables are fetched either from a relational database or read from flat files during DiceX launch time and then distributed among the data nodes

- In supercomputer's reservation based usage style, enable computations and analysis to be started from where it left off
- Prevent loss of updates to data made during the supercomputing session
- To help in preserving multiple versions of the same database and run evaluations on them whenever required

The idea is to store the `data home` directory of all the current data nodes in a dedicated storage server and associate a unique identifier with it. This can be considered as a file system level backup taken across all the current data nodes. When the distributed data engine is newly launched, instead of creating new data directories we could copy data directories from the storage server and resume operation. This will be helpful to continue working on the same data when a new supercomputing session is available to conduct experiments.

4.2 Use of MPI

MPI (Message Passing Interface) is chosen as a methodology to transfer data back and forth from the data nodes because it is known that it can transfer at a much faster rate when compared to traditional TCP. Previously, Zounmevo et al [57] implemented the network layer of a distributed HPC storage system using MPI to utilize the high speed communication, portability and performance offered by MPI. The following sections explain the MPI functions that were used and the experiments that were done in order to choose the best buffer size and number of MPI processes to be invoked on each compute node for the data transfer.

4.2.1 MPI Functions

Basic MPI Functions

- MPI_Send - Performs a blocking send from the source process to the destination process running in the same machine or in a different machine in the cluster.
- MPI_Recv - Performs a blocking receive of the message sent from the source process at the destination process running in the same machine or in a different machine in the cluster.
- MPI_Isend - Performs a non-blocking send from the source process to the destination process running in the same machine or in a different machine in the cluster.
- MPI_Irecv - Performs a non-blocking receive of the message sent from the source process at the destination process running in the same machine or in a different machine in the cluster.

- MPI_Wait - Takes a MPI request as input and waits until it is complete.
- MPI_Wtime - Returns time in seconds from an arbitrary time in the past. This method is used to time the data transfer operation.

MPI File IO

- MPI_File_open - Opens a file that can be shared among all the MPI processes in the provided communication object. It provides several file access modes including MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_CREATE, MPI_MODE_WRONLY, etc.
- MPI_File_seek - Updates file pointer to the given offset.
- MPI_File_read - Reads file contents into the provided buffer.
- MPI_File_write_at - Writes the contents of the provided buffer to the file referred by the given file pointer at the provided offset.
- MPI_File_close - Closes the file previously opened using MPI_File_open.

4.2.2 Experimental Results

Several experiments have been carried out to choose the optimum buffer size i.e. the chunk of data that should be sent during one MPI_Send and MPI_Recv cycle. These experiments were conducted by either reading from and writing to */localscratch* or */dev/shm* combined with a usage of blocking or asynchronous version of MPI send and receive. All the evaluations are presented in the sections below.

Transfer to and from */localscratch*

- Blocking MPI Send and Receive
  A simple blocking MPI send and receive model is used to read the data that is already archived into a tar ball and sent via MPI. A single sending process is invoked in the postgres-XC data node and a single receiving process is launched in the storage server where the data is archived. Figure 4.1 shows the time taken for various send and receive buffer sizes for transferring 22GB of data.

  Each experiment was run several times for the same buffer size and it has been found that the variance of the measurements is comparatively low and every measurement is always closer to a mean value for the same buffer size. Table 4.1 shows the mean and variance for the time taken from 5 experimental runs. We have conducted similar kind of runs for all the experiments presented in this thesis going forward and ensured that the values plotted in the graphs are always closer to a mean value.

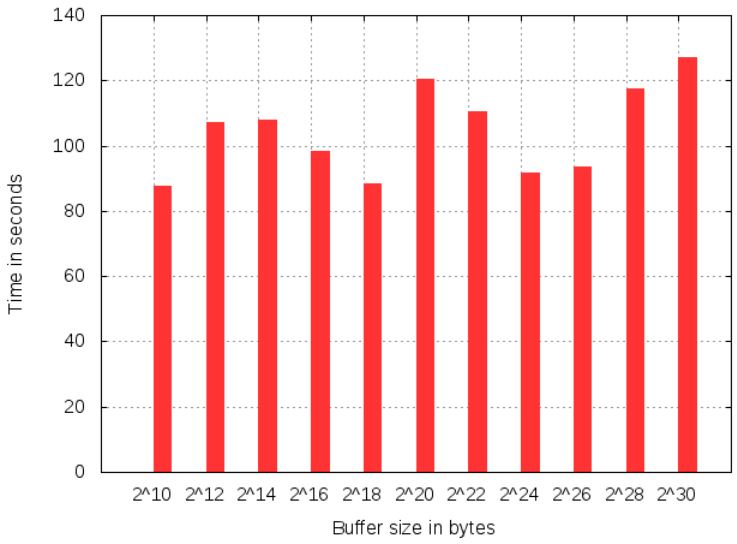


Figure 4.1: Time taken to perform blocking MPI I/O with one sender and one receiver both running on different nodes to transfer 22GB of data from */localscratch* for varying buffer sizes

Buffer size	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
Mean	88.12	107.1	108.05	97.92	87.91	120.53	109.77	91.86	92.7	119.7	127.62
Variance	1.38	1.45	2.67	1.55	1.23	1.54	0.44	0.76	0.85	1.11	2.54

Table 4.1: Mean and variance of measurements from 5 experimental runs for the same buffer size for blocking MPI I/O with one sender and one receiver to transfer 22GB of data from *localscratch*

- Asynchronous MPI Send and Receive
  Similarly asynchronous flavors of MPI send and receive functions were used with the same setup and figure 4.2 shows the time taken for varying buffer sizes.

It should be noted that a buffer size of 2^{18} i.e. 262144 bytes gives the best performance for both flavors.

Transfer to and from */dev/shm*

The same experiment performed on */localscratch* was carried out from */dev/shm* which is a temporary file system that uses RAM for its backing data storage. Since it uses RAM, it was anticipated that the data read and write could be faster. But, there is a limitation in the total size of */dev/shm* and so experiments were done only on a 4GB data sample.

- Blocking MPI Send and Receive

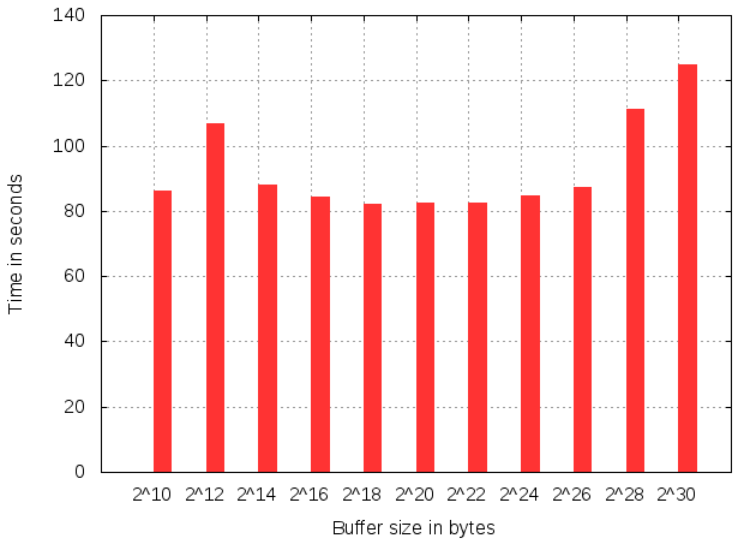


Figure 4.2: Time taken to perform asynchronous MPI I/O with one sender and one receiver both running on different nodes to transfer 22GB of data from */localscratch* for varying buffer sizes

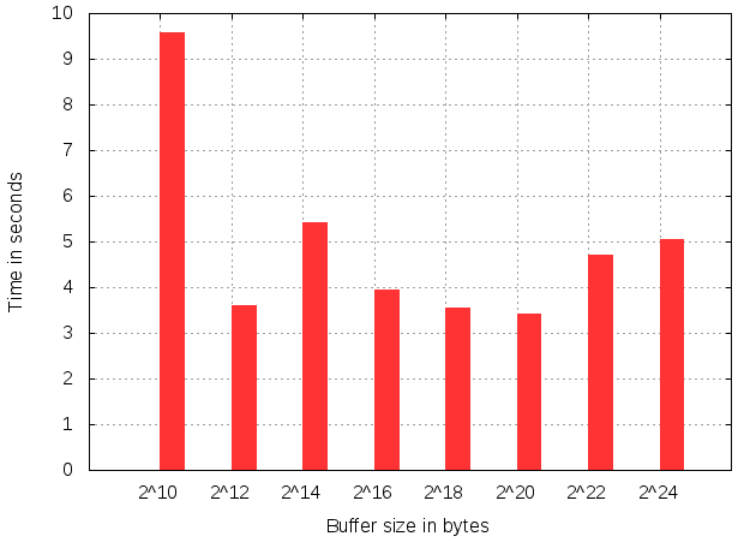


Figure 4.3: Time taken to perform blocking MPI I/O with one sender and one receiver both running on different nodes to transfer 4GB of data from */dev/shm* for varying buffer sizes

Figure 4.3 shows the performance of data transfer for varying buffer sizes while using */dev/shm*.

- Asynchronous MPI Send and Receive

Again, the asynchronous versions of MPI send and receive were tested on */dev/shm*

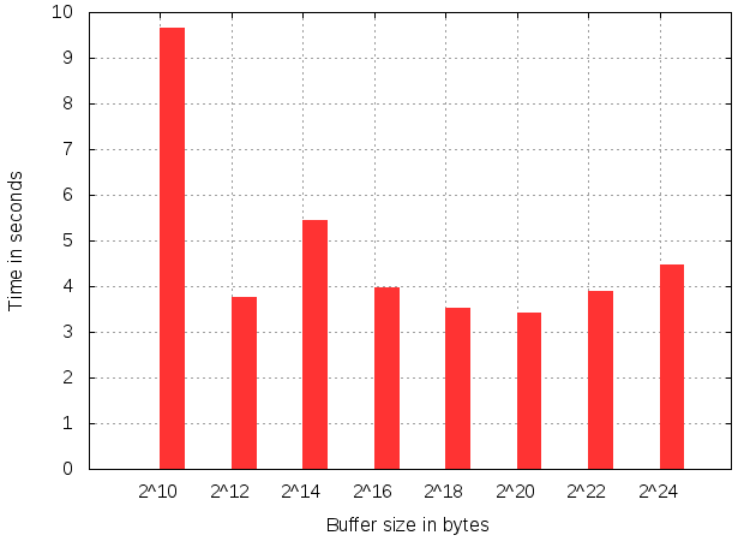


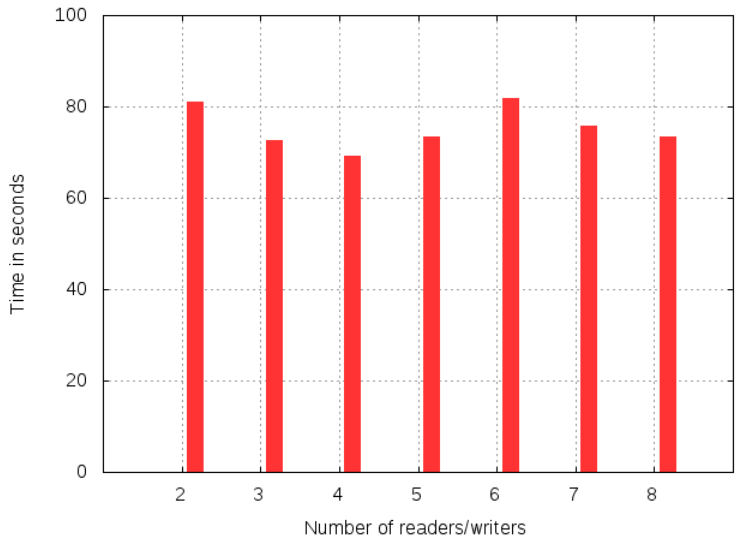
Figure 4.4: Time taken to perform asynchronous MPI I/O with one sender and one receiver both running on different nodes to transfer 4GB of data from */dev/shm* for varying buffer sizes

and Figure 4.4 shows the performance for varying buffer sizes.

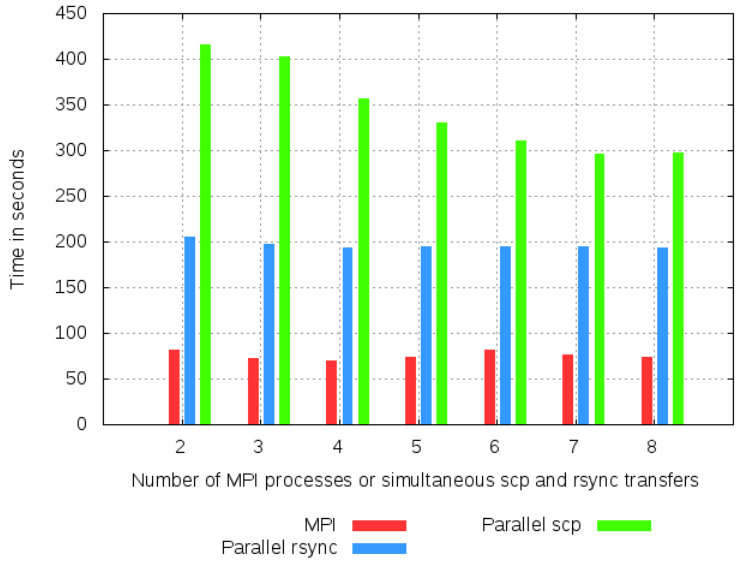
When using */dev/shm*, it is observed that using a send and receive buffer size of 2^{20} i.e. 1048576 bytes yields the highest transfer rate for both the MPI flavors. Since, */dev/shm* has a space limitation, it has been decided to utilize */localscratch* for storing the database image.

Multiple MPI Processes

In all the experiments listed above, only one MPI sender and receiver process was used for data transfer. In order to speed up the transfer rate, a preliminary experiment was conducted to select the optimum number of senders and receivers to be launched. Figure 4.5(a) shows the time taken for different number of sender/receiver pairs for transmitting 22GB of data using a buffer size of 262144 bytes. Data has been split equally among the specified number of senders and each sender would send its portion of data to the corresponding receiver process which would in turn do a MPI_File_write_at to write at the correct position without corrupting data. It can be seen that a model using 4 senders/receivers gives the best performance and therefore, it is chosen as the optimum number of processes to be used in the freeze & migrate and resume utility. It is suspected that maximum disk write capacity is reached with 4 simultaneous writing processes and anything beyond that would result in more contention and thereby produces a slower transfer rate.



(a) Performance of multiple MPI senders/receivers to transfer 22GB of data using a buffer size of 262144 bytes



(b) Comparison of MPI with parallel rsync and parallel scp

Figure 4.5: Multiple MPI senders/receivers vs Parallel rsync vs Parallel scp

Figure 4.5(b) compares MPI with parallel `rsync` and parallel `scp`. It can be seen that MPI performs much better than both `rsync` and `scp` for the same number of simultaneous transfers. Increasing the number of parallel `scp` or `rsync` transfers doesn't help much in improving the speed of data transfer. Though the time taken by parallel SCP is reduced, the reduction isn't big enough to make an impact and in the case of parallel `rsync` transfers, the

reduction in time taken was negligible. Nevertheless, parallel `rsync` is faster than parallel `scp` as anticipated.

Since, we had fixed 4 MPI readers/writers to be used in the freeze & migrate and resume utility service, a final experiment was conducted to verify that the buffer size of 262144 bytes does indeed gives the best performance. Figure 4.6 shows the performance of using 4 sender processes and 4 receiver processes on varying buffer sizes while transferring data of size 22GB and it does confirms that 2^{18} i.e. 262144 bytes is the best buffer size for maximized performance.

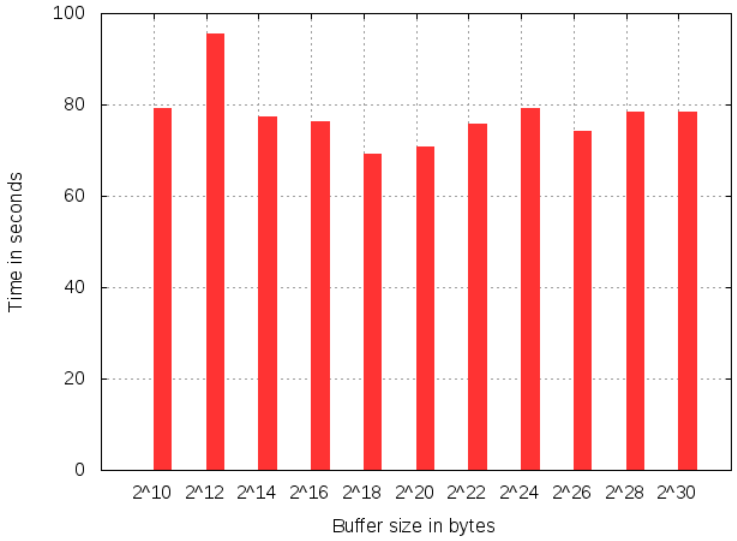


Figure 4.6: Time taken to perform blocking MPI I/O with four senders and four receivers to transfer 22GB of data from */localscratch* for varying buffer sizes

4.3 Freeze & Migrate and Resume Procedure

From the above experimental results, it has been identified that having 4 MPI readers and writers using a buffer size of 262144 bytes gives the best performance to transfer data. At the end of a supercomputing session or whenever there is a requirement to checkpoint and preserve an image of the current data, the DiceX freeze & migrate process can be invoked which takes care of archiving the data directories into `.tar` files and perform a MPI data transfer to the dedicated storage server. The database image is given a unique identifier called VERSION_ID which is of the form `date_timestamp`. Whenever DiceX needs to be launched again with the preserved data, this VERSION_ID is used to uniquely identify the image that is of interest. The working schematics of DiceX freeze & migrate and resume utility service is shown in figure 4.7.

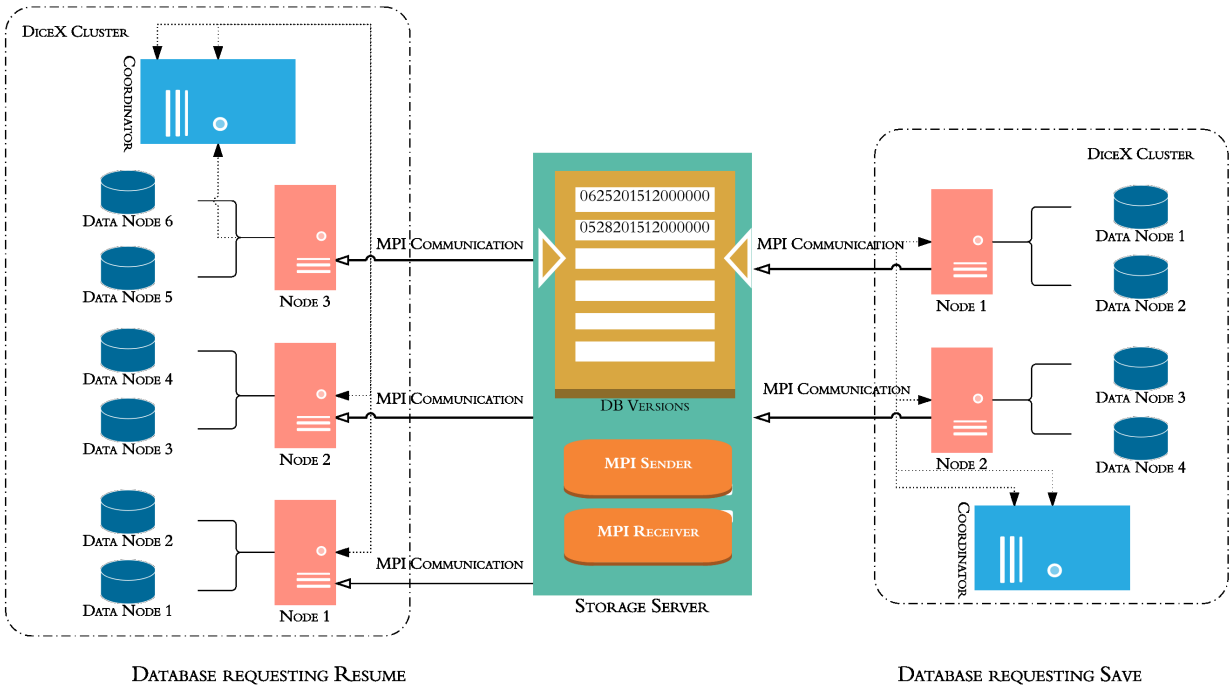


Figure 4.7: Freeze & Migrate and Resume Schematics

Figure 4.8 shows the time taken for freezing and migrating a database image of size 19GB that is distributed among a varied number of datanodes. The time taken to launch DiceX with the saved image using the resume utility is also shown in the same graph. It is observed that the time taken to resume from image is almost uniform for varying numbers of data nodes.

As mentioned in the beginning of the chapter, DiceX offers optimized data ingestion to a table from a file or from a remote database by partitioning the data source and ingesting each partition concurrently [37]. Figure 4.9 compares the time taken for launching DiceX: without any data; with parallel data ingestion from a relational database(Oracle) using 12 partitions; with data ingested through COPY command; and with the resume service. It can be seen that the time taken by the resume utility isn't considerably higher when compared to launching DiceX without any data. Current multi view parallel ingestion of data offered by DiceX is no way near the performance of resume service. Data ingestion through COPY command from flat files performed better than parallel data ingestion but it is worse than our resume service. Moreover, for large datasets, it is expected that the resume service should be many times better than all other data ingestion methods.

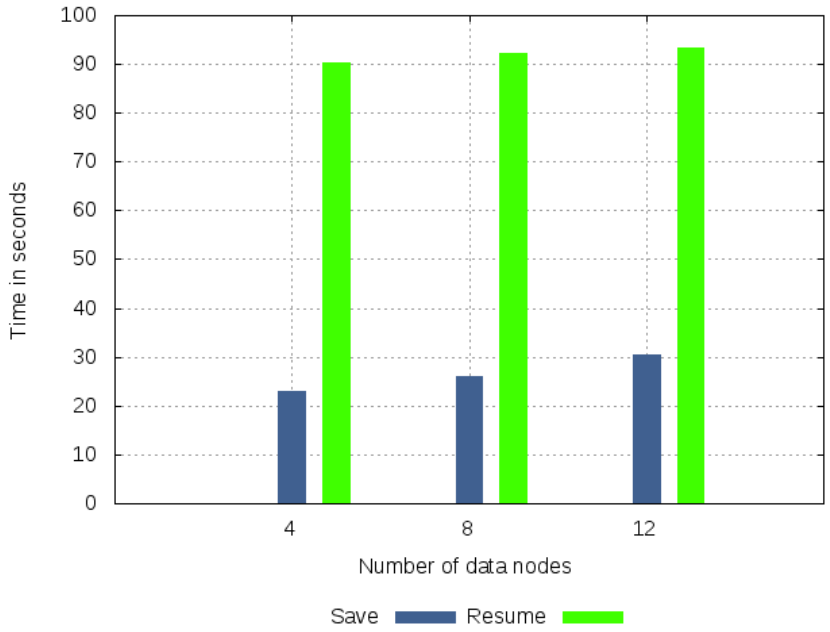


Figure 4.8: Time taken to Freeze & Migrate and Resume 19GB of data for varying number of data nodes

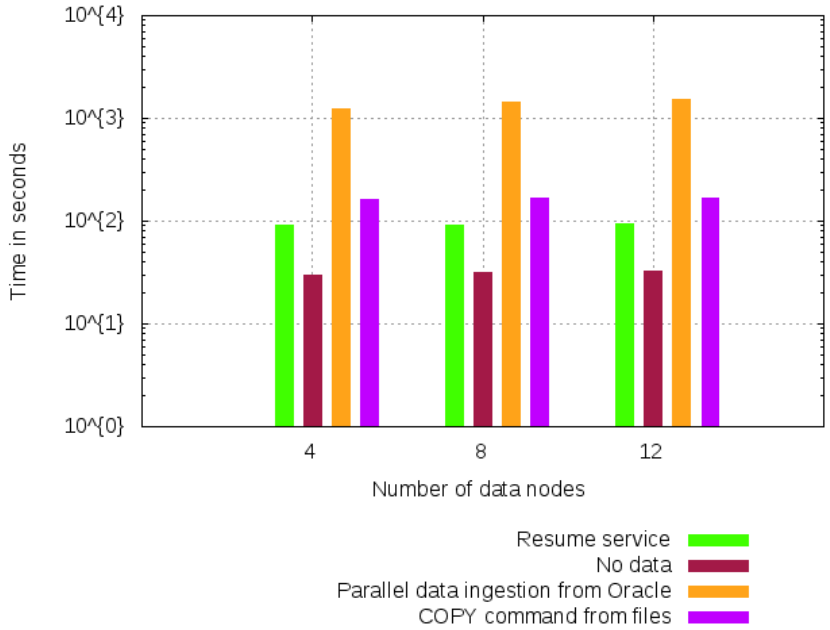
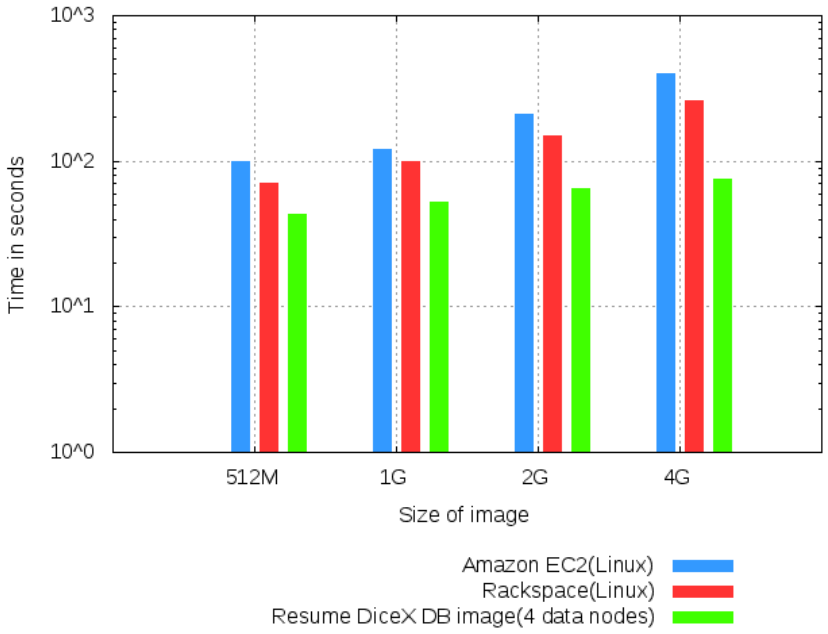
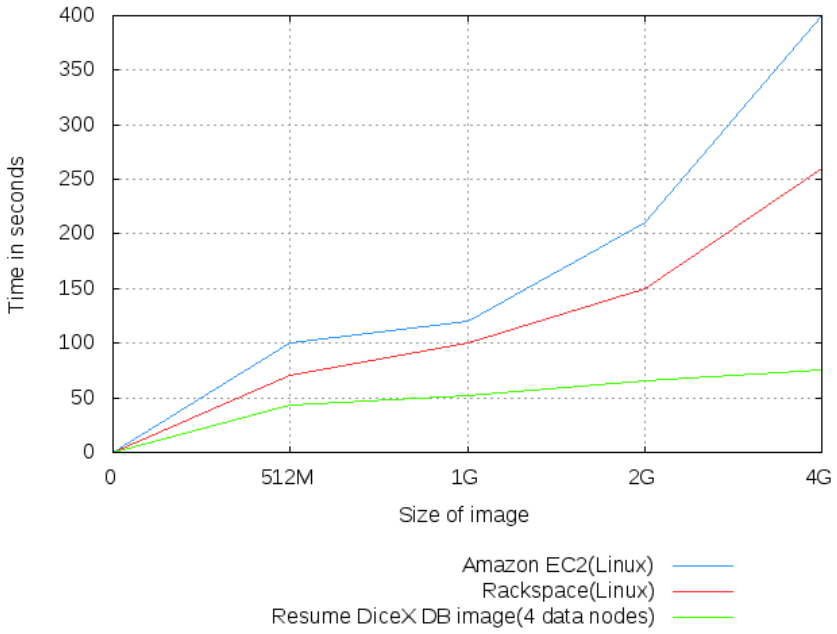


Figure 4.9: Comparison of time taken for launching DiceX with resume service, with no data, with parallel data ingestion from Oracle, with COPY from files for data of size 19GB

4.3.1 Comparison with cloud services VM startup time



(a) Comparison in log scale



(b) Comparison in linear scale

Figure 4.10: Comparison of time taken to load from database image using resume service with load from VM image by popular cloud service providers

The database freeze & migrate and resume procedure is analogous to how cloud computing services would preserve the image of the VM (Virtual Machine) at the end of a session and load it from the saved image upon new login. [41] compares the VM startup delay in popular cloud service providers for loading a VM image of a Linux operating system for varied image sizes. We compared the time taken by our database resume service to restore the database image for the same image sizes in figure 4.10. It should be noted that our resume service performs several magnitudes better than Rackspace which is the best among the two cloud service providers.

4.4 Other Use cases

As mentioned before, one of the major motivations behind adding this feature to DiceX is to reduce the parallel database start-up time. We believe that this utility can also be immensely helpful in the following scenarios:

- A scientist working on designing an intervention strategy for a disease where the results of one iteration is dependent on the previous iteration can make use of the utility to store the previous iteration's data.
- With the freeze & migrate and resume utility, a scientist who is using DiceX can take snapshots of the data that they had been working on and restore from any of the snapshot with little or no time cost.
- Repeated experiments on the same data can be easily done, as the same saved version can be launched multiple times. A scientist working on certain machine learning algorithm can run an algorithm on the same test data with varied parameters.
- Every time the resume service is invoked on a particular saved version, it will launch onto a separate set of supercomputing nodes. This will help a scientist in launching the same version of test data multiple times and run an algorithm with varied parameters concurrently, thereby helping in simultaneous comparisons and tuning of parameters.

Chapter 5

DiceX - Cluster Elasticity

DiceX currently does not support the concept of elasticity, i.e. dynamic ad-hoc extension of data nodes as and when required. This is facilitated by our new node add utility. Once, the database engine is launched onto the supercomputer, this utility helps in extending the number of data nodes and redistributing the tables without having to stop any running services. With this feature, we address the challenge of re-sizing the data processing engine that we launch onto the supercomputer.

5.1 Motivation

The primary motivation behind implementing this Node add utility to DiceX is to achieve the following goals:

- It is not always possible to determine the correct number of data nodes to attain maximum performance before launching the parallel database using the DiceX launch utility. In order to improve performance, the number of data nodes may have to be increased and currently the only way to achieve this is to relaunch DiceX with the new number of data nodes and reload all the required data.
- The requested number of compute nodes may not always be available at the time it is requested. This ad-hoc node addition utility assists in extending the number of data nodes as and when compute nodes become available in the cluster. This enables a 'slow start' approach to the framework.
- When a table grows in size due to frequent row insertions (and even though we have hash distributed the table among the current set of data nodes), addition of a new node and redistributing the table data can help in reducing the running time on the data nodes in some cases, as this downsizes the number of records at each data node.

With the above use cases, cluster elasticity also helps in scaling large datasets over commodity hardware on an ad-hoc basis. The concept of cluster elasticity has been popular with cloud services but it has always been believed that it would be hard to achieve in relational databases. DiceX can extend its resources with ease and without having to cease operation.

5.2 Node add utility

Since, a new data node is added to a cluster that is already up and running, a number of alterations must be made to the current configuration. DiceX node add utility accomplishes this by executing a script and specifying the required number of new compute nodes, the number of data nodes that should run on each of these new compute nodes, and the tables to be redistributed.

5.2.1 Behind the scene

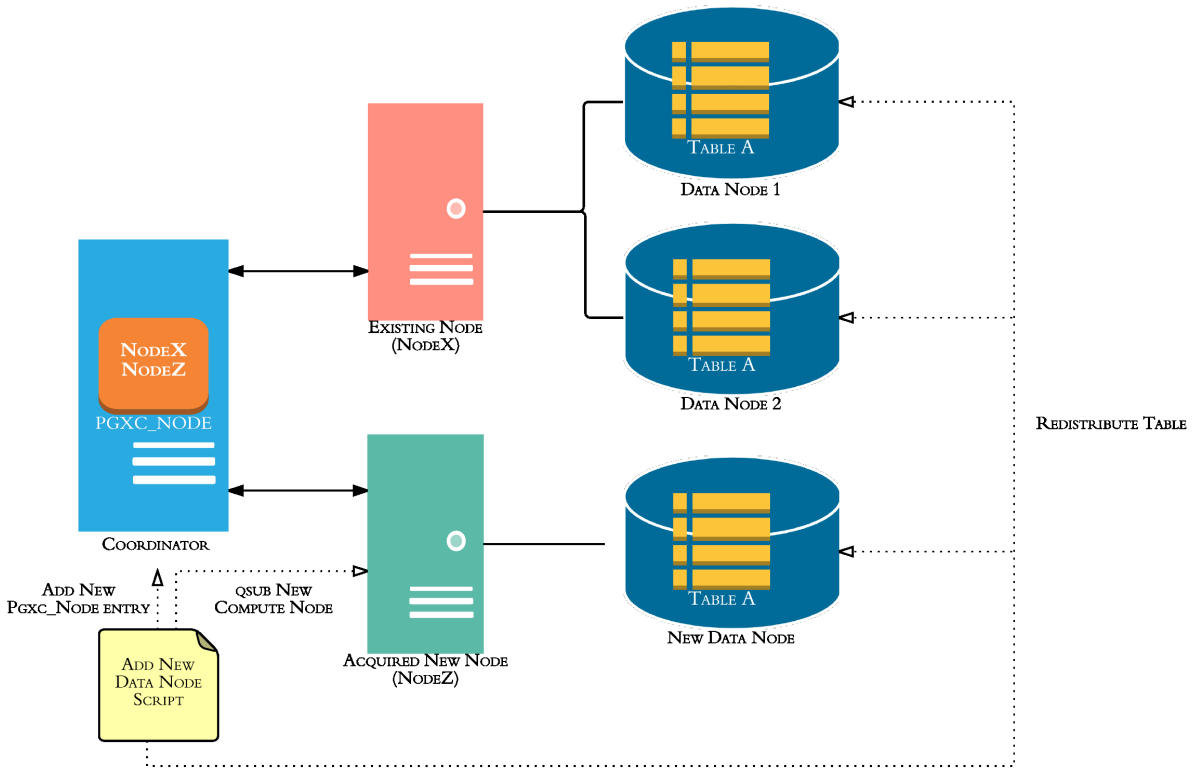


Figure 5.1: Schematic of node add utility

Figure 5.1 shows the basic schematics of the functioning of the node add utility. This utility does the following behind the scenes to include new data nodes in the current postgres-XC cluster:

- Launch `qsub` jobs to acquire the given number of new compute nodes from the supercomputer
- Initialize given number of data nodes and create empty data directories on the acquired compute nodes
- Lock the current postgres-XC cluster for backup so that no change to the schema is issued
- Connect to one of the existing data nodes and issue `pg_dump` for getting a backup of the schema alone
- Start the data node service in the new nodes in restore mode and restore from the schema backup created in the previous step
- Quit the data node service and restart in normal mode
- Unlock the postgres-XC cluster
- Connect to the coordinator and do the following:
  - Create new `pgxc_node` table entries for the new data nodes
  - Issue `pgxc_pool_reload()` to reflect the addition of new data nodes at the coordinator pooler process
  - Issue `alter table` commands on the tables to be redistributed

5.3 Evaluation

In order to evaluate the new node add utility, diverse experiments were conducted and its performance analysed. It is critical that the DiceX node add utility should not take more time to extend the number of data nodes in the cluster and redistribute the required tables. A test environment is set up at **NDSSL shadowfax** cluster and DiceX launched with a total of 4 data nodes where 2 data nodes were launched on one compute node. A dataset containing demographic information for the state of Texas with 22000483 rows of data is used for these evaluations.

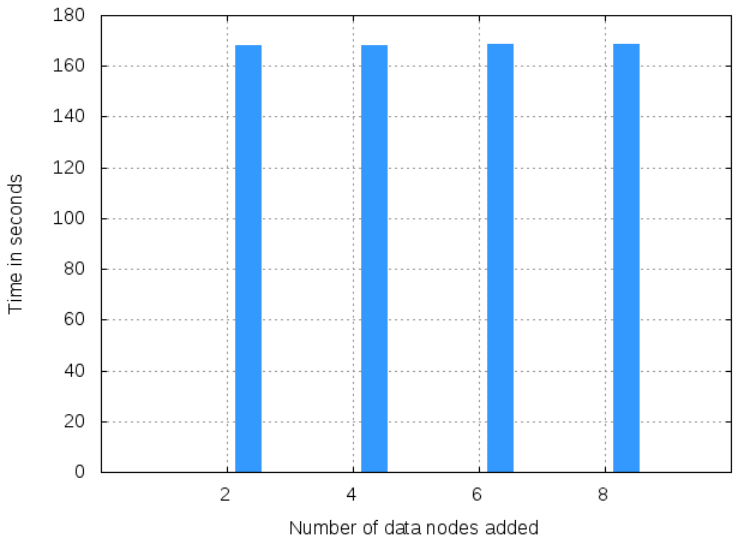


Figure 5.2: Time taken to add new data nodes and redistribute a table with 22000483 rows for varying number of new data nodes that are added

5.3.1 Hash distributed table redistribution

For this experiment, the dataset was hash distributed among the 4 data nodes. The number of data nodes was extended in increments of powers of 2. Figure 5.2 shows the time taken to add varied numbers of new data nodes to the original cluster and redistribute the tables to the extended cluster.

It should be noted that the difference in time taken to add 2 data nodes and 8 data nodes is negligible. Therefore it is claimed that the node add utility scales well.

5.3.2 Replicated table redistribution

The same experiment was conducted by replicating the table to the newly added data nodes instead of hash distributing it. Figure 5.3 shows the time taken to add new data nodes and replicating the table to these newly added data nodes. It is obvious that the time taken to replicate is higher than hash distributing it, as the entire table data has to be replicated in all the new data nodes.

5.3.3 Redistribution vs fresh ingestion from file

DiceX node add utility redistributes the table among the new data nodes by issuing the modified `alter table` command introduced with postgres-XC. This redistribution process

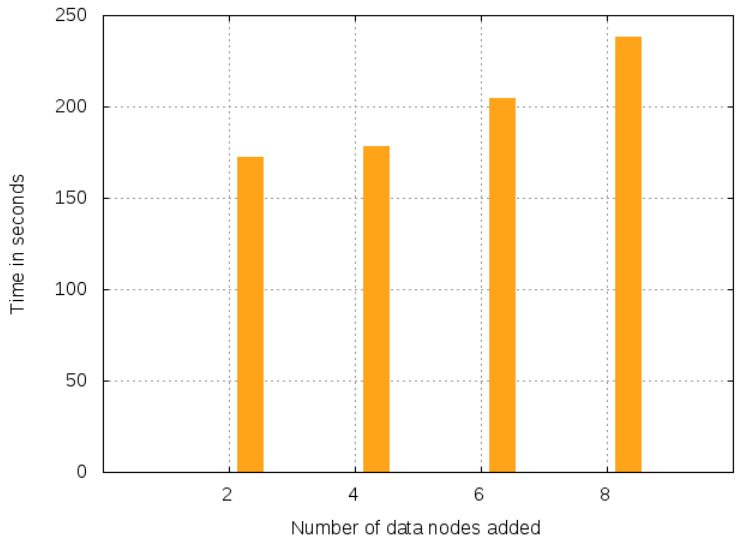


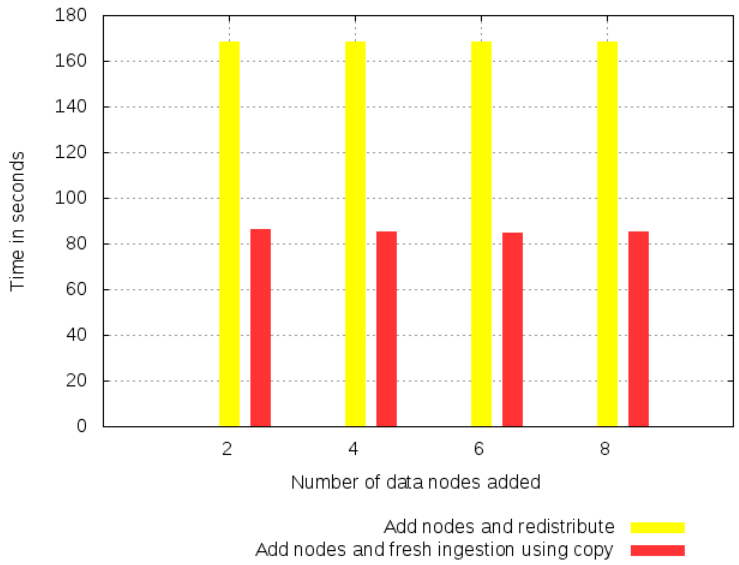
Figure 5.3: Time taken to add new data nodes and replicate a table with 22000483 rows for varying number of new data nodes that are added

of postgres-XC happens as follows:

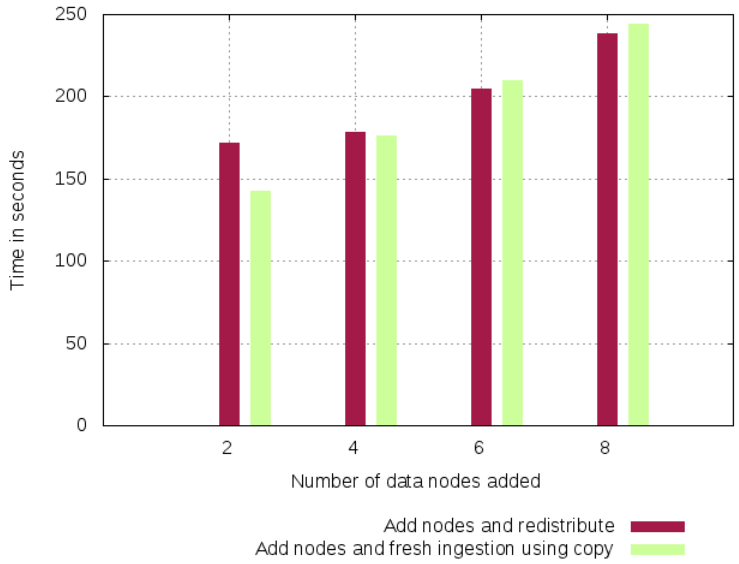
- Fetch all data of the table that needs redistribution at the coordinator's `Tuple Store` using a `copy` protocol
- Truncate the existing table at all the data nodes
- Update the distribution type of the table at the coordinator catalogs
- Redistribute the cached data at the coordinator to all the data nodes including the newly added ones

A separate experiment was conducted to compare the time taken to redistribute the table among the data nodes and ingesting data from a file using SQL's `copy` command after adding the new data nodes. Initially, the Texas demographic dataset was loaded onto 4 data nodes either by hash distribution on the primary key or by replication and the DiceX node add utility was used to add new data nodes. Figure 5.4 shows the comparison results of this experiment. It should be noted that when the table is hash distributed, `COPY` command performs far better than table redistribution. Even though redistribution is slower, it is within the acceptable limits and there is scope for improvement. In the case of a replicated table, when the number of data nodes being added increases, table redistribution performs slightly better than `COPY` command.

Therefore, it is advisable to resort to fresh ingestion using `COPY` command when the table



(a) Hash distributed table



(b) Replicated table

Figure 5.4: Comparison of table redistribution vs fresh ingestion from file using COPY command for a table with 22000483 rows for varying number of new data nodes that are added

is hash distributed. When the table is replicated, either of these methods can be applied as they both take almost the same amount of time.

5.3.4 Changing distribution column

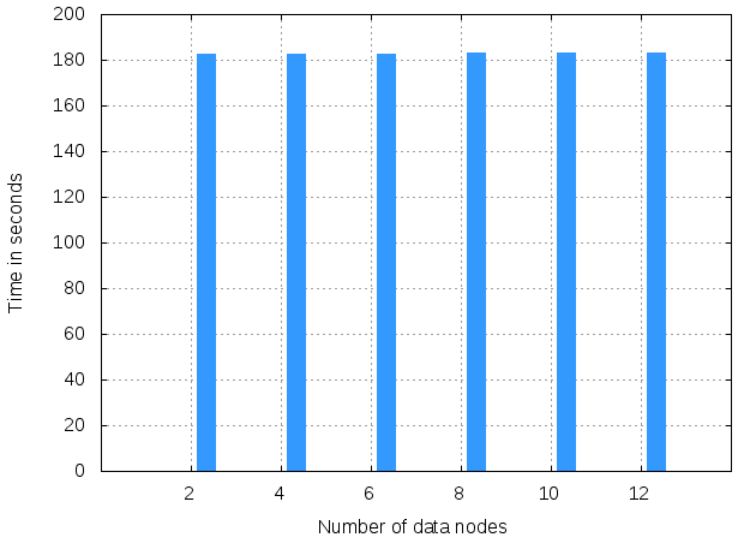


Figure 5.5: Time taken to change distribution column of a table with 22000483 rows for a varying number of data nodes

Figure 5.5 shows the time taken to change the distribution column of a table with over 22 million rows that has already been hash distributed by a different column for a varied number of data nodes. We observe almost constant running time to change the distribution column while varying the number of data nodes. This is advantageous because it is easy to change the distribution column when required to aid in pushing down time consuming SQL operations to the data nodes (as we see in chapter 6) and later revert to the original distribution column.

We conducted a separate experiment to find the time required to change the distribution column for tables of varying sizes on a fixed number of 8 data nodes with 2 data nodes per compute node. Although, time taken to redistribute is within the acceptable limits, it can be seen from the figure that it isn't the best option if a many tables with large numbers of rows must be redistributed. Fresh ingestion using COPY command would perform far better.

5.3.5 Changing distribution type

Apart from adding new data nodes and redistributing the table in the same mode in which the table was created, it is also possible to change the distribution type of the table upon adding new data nodes. Figure 5.7 shows the time taken to change the distribution type of the table for a varying number of data nodes. Both changing from hash distribution to replication and vice versa have been analysed. Note that the plot only shows the time

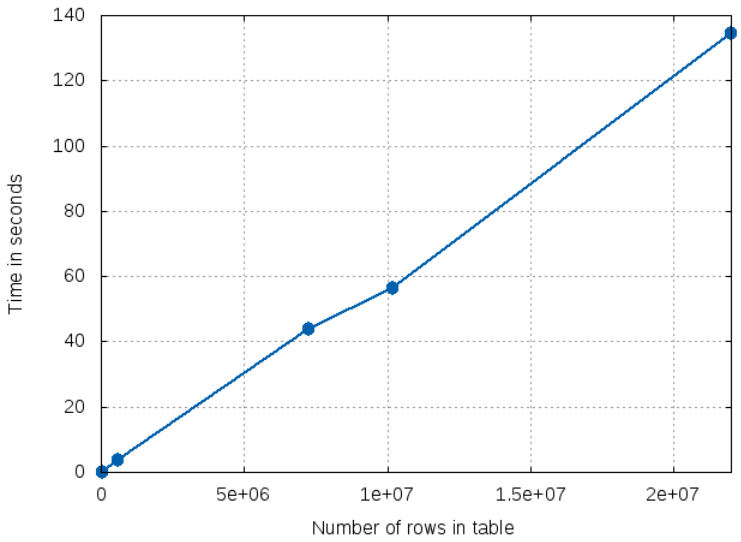


Figure 5.6: Time taken to change distribution column of a table for varying table size on 8 data nodes

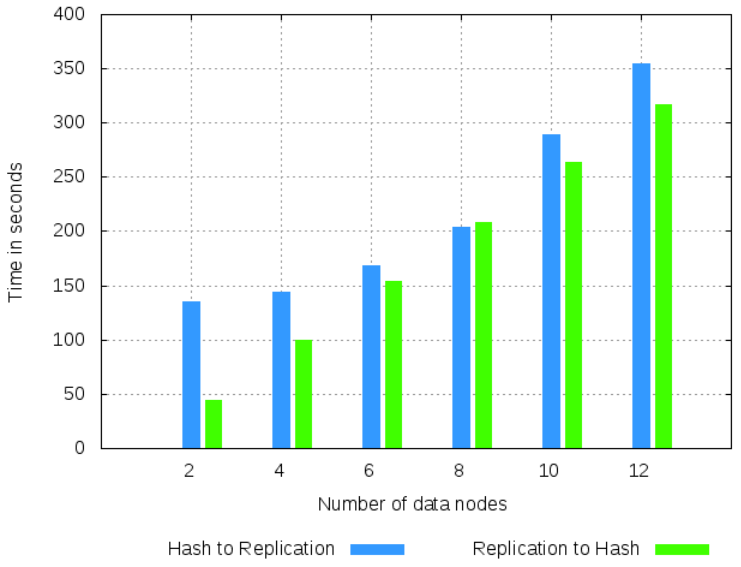


Figure 5.7: Time required to change distribution type of a table with 22000483 rows for a varying number of data nodes

required for changing the distribution type of the table and doesn't include time required for adding new data nodes, as only changing the distribution type is of interest for this study.

Even though it is believed that a replicated table doesn't aid in parallel execution of queries, it can be of use when an application demands de-normalization to provide high availability.

Also, it is easy to switch back to hash distribution without having to stop any service.

Chapter 6

Postgres-XC Optimizations

In this chapter, the optimizations that had been performed on our distributed data processing engine, postgres-XC, is elucidated in detail. DiceX query processing is done at both the data nodes and the coordinator and it is important to analyse how a SQL query is processed in postgres-XC. Deeper introspection on the internal query processing will help in getting a better understanding of the bottlenecks enforced by the postgres-XC distributed database system as well as ways to further improve its performance. Maximizing parallel execution of the operations can help in achieving faster execution rates of even complex queries.

6.1 Postgres-XC terminology

Before analysing the internals of query processing in postgres-XC, the following terminologies should be understood.

- `quals`
  The expressions or conditions listed in the WHERE condition of a SQL query.
- `clause`
  The operations like `join`, `group by`, etc., that are present in the query.
- `ExecNodes`
  The list of data nodes that are involved in query processing.
- `RemoteQuery`
  The query that will be executed at the data nodes.
- `RemoteQueryPath`
  This represents the output of the postgres-XC planner i.e. all the remote queries that will be sent to respective data nodes for execution.

- `TupleSlot`
  The tuple object onto which the raw data is written in a data row format at the coordinator.

- `TupleStore`
  `TupleStore` is the data structure that holds the resultant rows of a query at the coordinator. The raw data from the `ExecNodes` are written in a data row format into the tuple slots which are members of the `TupleStore`. When the maximum allowed size for a `TupleStore` is reached, it is written to a file at the coordinator to accommodate space for more data that is yet to be received from the `ExecNodes`.

6.2 Query push down and data distribution

We describe a query as pushed down when it is entirely evaluated at the data nodes for processing. In this case, the coordinator simply aggregates the results from the data nodes. Otherwise, the query is only evaluated partially at the data nodes i.e. only a subset of the query's clauses are evaluated on the data nodes. The coordinator evaluates the rest of the query using the intermediate results from the data nodes.

In some trivial scenarios, the data nodes merely perform a scan of their local table data and send the results to the coordinator. Consider the following simple query with a join operation,

```
SELECT
        COUNT(a.order_id)
FROM
        Table1 d, Table2 a
WHERE
        d.order_id=a.order_id;
```

If either both the tables are present in different set of data nodes or if they are hash distributed among all the data nodes but by a different column other than `order_id`, then all the data nodes will simply scan their data and send to the coordinator. The `join` operation along with the `count` aggregate operation is performed at the coordinator. This results in poor query performance and will probably perform worser than standalone postgreSQL.

Finally, in some scenarios, when a query is complex and require multiple steps of processing and at every step, a fraction of the complex query is processed at the data nodes and the coordinator before proceeding to the next step. Consider the example:

```
SELECT
        t.invoice_id
```

```
FROM
        (SELECT
                count(*) as total, invoice_id
        FROM
                Table1 d, Table2 s
        WHERE
                d.item_id = s.item_id
        GROUP BY
                d.invoice_id, d.order_id) INTERMEDIATE, Table3 t
WHERE
        t.invoice_id = INTERMEDIATE.invoice_id;
```

Here the query has a sub query that must be completed first and so this query involves multiple steps of execution at the data nodes and coordinator. The data nodes will first execute the portion of the sub query that can be done in parallel and send their intermediate results to the coordinator. The coordinator will then complete the remaining portion of the sub query, i.e. the part that cannot be executed in parallel and has to happen at the coordinator. Here the `count(*)` operation of the sub query is performed at the coordinator to get the `total`. Upon completion of the sub query, the remaining part of the query execution continues at the data nodes and coordinator. This is one simple example of multi step query execution. Some complex queries can require many rounds of query processing at the data nodes and coordinator to arrive at the final results.

The performance of a query is strongly determined by the distributed query plan in all these scenarios. Generally, a push down query is faster than a partially pushed down query which is in turn faster than a multiple step evaluation query.

The postgres-XC database planner decides to push down a query either partially or completely based on the distribution of tables present in the query and their associated clauses. The list of clauses and data distribution combination that the planner adopts for pushing down operations is discussed in the sections below.

6.2.1 Criteria for quals

Quals in the query are almost always shipped down to data nodes along with the other operations but are restricted on certain occasions. For example, quals calling other mutable functions are always executed at the coordinator. Consider the SQL query:

```
SELECT
        order_date
FROM
        Table1
```

```
WHERE
        order_date < CURRENT_DATE - 5;
```

Here the call to a mutable function `CURRENT_DATE` will prevent the filter condition from being shipped down to the data nodes.

6.2.2 Criteria for JOINs

Join operations are handled separately by the planner and several conditions determine whether to ship it down to the `ExecNodes` or not. If it is determined that the join operation cannot be pushed down, it is performed at the coordinator. In case of inner joins all the `quals` other than the joining condition are also shipped down whereas in case of outer joins only the joining condition is pushed down and all the other `quals` are executed at the coordinator. Join operations are pushed down to the data nodes based on the following conditions [11]:

- The join operation is either inner, left outer or full join. Right outer joins are not pushed down
- Either both the inner and outer relations involved in the join are replicated tables or distributed by the joining column
- If the outer relation is distributed and the inner relation is replicated only left outer and inner joins are pushed down
- If the outer relation is replicated and the inner relation is distributed, only inner join is pushed down to the data nodes

6.2.3 Criteria for GROUP BYs

Similar to joins, group by operations are also pushed down to improve parallelism and reduce the amount of data that is being sent to the coordinator. Again, group by operations are pushed down based on certain governing conditions [11].

- Group by is shipped down only if it contains plain aggregates or expressions. Group by with aggregates will be pushed down if it involves only one data node or the group by is on the distribution column
- If the SQL statement has un-grouped columns which are functionally dependent upon grouping columns, then group by operation cannot be pushed down
- When an ORDER BY clause accompanies a group by, then the operation cannot be pushed down

- When a DISTINCT clause or HAVING clause accompanies a group by, then the operation cannot be pushed down unless there is only one data node that is involved or the group by is on the distribution column

6.2.4 Data distribution aiding push down

In this thesis, the above mentioned shippability criteria are exploited to determine the best data distribution strategy for tables that would yield maximum push down of the queries and performance scaling.

For example, since joins and group bys are pushed down only if they are performed on the distribution column, the queries that are used in epidemiological research are carefully studied and the data is hash distributed among the data nodes using the column on which these operations are performed the most. From the queries given in Appendix B which are used for evaluation, it is understood that the `pid` column which corresponds to the unique id assigned to a person in the dataset is used for all the join operations. So, it is the most obvious choice to be used for distributing data among the data nodes. Also, mutable functions that involves time and date manipulation used in the query are avoided and instead are precomputed and converted to numbers before querying. Queries q4, q6, q7, q10, q12 are good examples where we had precomputed the date operation and simply fed the numbers for the range condition on the `exposed_time`.

6.2.5 Using domain semantics to force query push down

Several queries in epidemiology workloads cannot be completely shipped down to the data nodes. Typically these queries have some SQL operations like `join`, `group by` over two different attributes. Therefore, a fraction of the query which has the clause over the distribution column can be pushed down but the remaining cannot. Lets look at two example scenarios that help to understand this limitation.

Example 1

Consider the example schema given in figure 6.1 which shows how an epidemic infection data could be stored in relational tables. Let's assume that these tables are hash distributed by the column `COUNTY_ID` to all the data nodes. Consider that when the following query, which is looking for the total infection count of the epidemic at the block group level of the state is submitted for execution, not all the operations are pushed down to the data nodes.

```
SELECT
        COUNT(i.PERSON_ID), d.BLOCKGROUP_ID
```

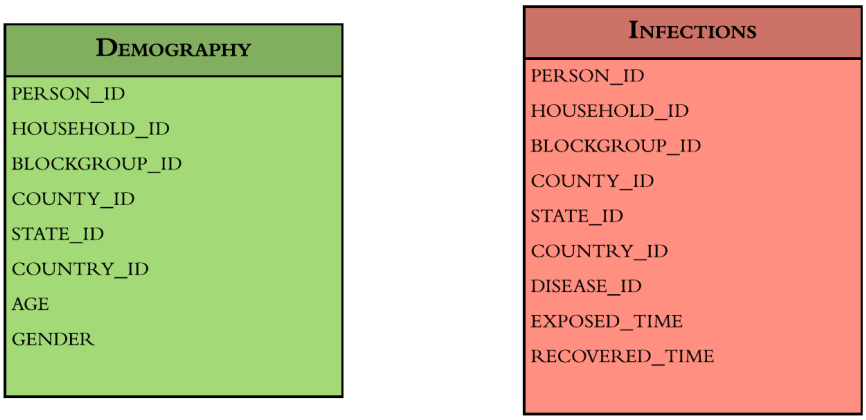


Figure 6.1: Example epidemic infection relational schema

```
FROM
        DEMOGRAPHY d, INFECTIONS i
WHERE
        d.PERSON_ID = i.PERSON_ID AND
        d.STATE_ID = 'Virginia'
GROUP BY
        d.BLOCKGROUP_ID;
ORDER BY
        d.BLOCKGROUP_ID DESC;
```

This is mainly because of the shippability restrictions enforced by postgres-XC which prevent some operations from being pushed down. In this query, the `join` operation cannot be pushed down as it is not performed on the distribution column. Similarly, the `group by` and `order by` operations are also not pushed down based on the same condition. The only operation that will be pushed down is the `where` clause condition `d.STATE_ID = 'Virginia'`.

If the tables were distributed by the column `PERSON_ID`, then the `join` operation would be pushed down but it is not always possible to assume that all the queries will be performing joins on the same column `PERSON_ID`. Queries from computational epidemiology more often perform joins on more than one column and a more generic solution is required to completely push down the queries.

Example 2

Consider the same example schema from figure 6.1 and now let us assume that the tables have been hash distributed by the `PERSON_ID` column to all the data nodes. If the following

query is executed to find the block group that has the maximum infection in a county, again, not all the operations of the query are pushed down because of shippability restrictions.

```
SELECT
        COUNT(i.PERSON_ID), d.BLOCKGROUP_ID
FROM
        DEMOGRAPHY d, INFECTIONS i
WHERE
        d.PERSON_ID = i.PERSON_ID AND
        d.STATE_ID = 'Virginia'AND
        d.COUNTY_ID = 'Montgomery'
GROUP BY
        d.BLOCKGROUP_ID
ORDER BY
        d.BLOCKGROUP_ID DESC
LIMIT 1;
```

Now, the join operation along with all conditions in the `where` clause will be pushed down to the data nodes but the `group by` and `order by` operations cannot be pushed down because of the shippability criterion that they are not by the distribution column. Even if they are pushed down, it is not assured that all the records corresponding to a particular block group end up in the same data node and therefore another group aggregation must be performed at the coordinator. The `limit` operation cannot be pushed down and should always happen at the coordinator. Even though, this query pushes most of its operations to the data nodes, a careful assessment of the data semantics could have led to a better distribution of data to enhance majority operation push downs.

The above two examples show that data distribution can enhance as well as reduce the number of SQL operations being push down to the data nodes. In the first example, if the tables were distributed by the `PERSON_ID` column, it could have helped in pushing down all the operations, whereas in the second example data distribution by `PERSON_ID` prevented pushing down of some operations. Thus, a proper well educated data distribution can help in enhancing the parallel execution of SQL operations in a query.

New push down criterion

We propose a new push down criterion that is based on the semantics of the domain under study. With this criterion, queries such as those above can be pushed down completely to the data nodes. However, for the query result to be correct, it requires the data to satisfy the following relationship constraint.

One-to-many relationship: In general relational database terminology, an one-to-many relationship is identified when one single row of the related table is associated with more

than one row of the relating table. But, in epidemiology datasets such relationships can exist even within a single table among the columns of the table. This one-to-many relationship can help in pushing down operations even on columns that haven't been used to distribute data. This is because when data is distributed on the column which has a one-to-many relationship with the join by column, it can be claimed that all the required data for the join operation will end up being available in the same data node.

Such constraints can only be defined by understanding the semantics of the relationship between the attributes at the domain level. For example, in the domain of computational epidemiology, the following one-to-many relationship exists:

- A state can contain many counties
- A county can contain many block groups
- A block group can contain many households
- A household can contain many people
- The same person can be infected and recover multiple times during the epidemic season

All these are one-to-many relationships, a form of containment which is common to epidemiology analysis. Understanding them can help in devising an efficient method to distribute table data that can aid in achieving maximized performance without compromising correctness.

Engineering force push down

To implement force push down in postgres-XC, its default query planning and shippability criteria have been studied carefully and appropriate changes have been made to its shippability module. In particular, while determining shippability of joins and group bys, an extra condition has been added to see if there is a file in the name of the join column or group by column in the coordinator's data directory. If the file is present, irrespective of the outcome of the default shippability module, the operations are forcefully pushed down to the data nodes. Thus, it is essential for the researcher to place files in the names of the columns that they would often perform joins and group bys and would want to bypass the default shipping criteria. This is more of an informal approach to engineer force push down in postgres-XC and a more formal technique can be devised in the future like storing the columns in a separate table in the database and check with that table entries before determining whether to force push down SQL operations.

The domain based force query push down has been implemented in postgres-XC and succeeded in reducing the computational time of several queries. For this experiment, the

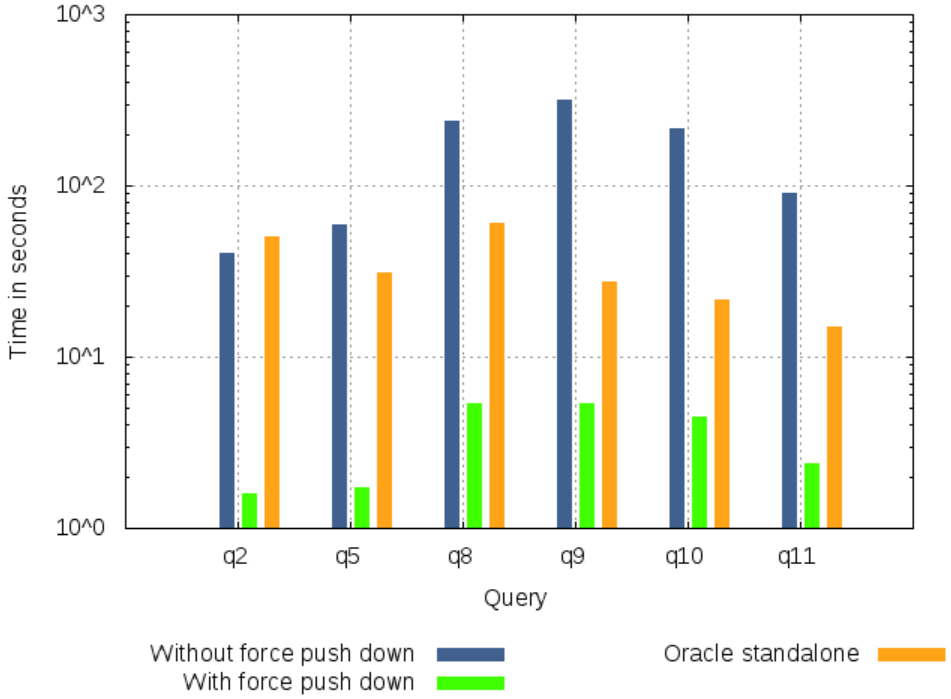


Figure 6.2: Comparison of query performance with force push down turned on vs without force push down when data is hash distributed by `blockgroupid` vs standalone Oracle database

demographic and infection table from schema given in Appendix A are hash distributed among 12 data nodes running on 6 compute nodes by the `blockgroupid` instead of the column `pid` on which the join operations are performed. Basically, in the queries shown in the experimental results, joins and group bys are forcefully pushed down avoiding the restriction criteria enforced by postgres-XC, because the data distribution based on `blockgroupid` will not affect the correctness of joins on `pid` because data of people belonging to the same block group are guaranteed to be available in the same data node because of the one-to-many domain semantics between block groups and people. Figure 6.2 compares time required with and without the domain based force query push down in postgres-XC with Oracle standalone database on certain queries from Appendix B.

As can be seen for some queries, the Oracle standalone version performed better than postgres-XC without force push down because of the extra overhead in distributed query execution and the inability to perform time consuming `join` and `groupby` operations on the data nodes. Whereas, the performance of postgres-XC with force push down produces a 40%-60% reduction in the time taken to execute these queries when compared to both postgres-XC without force push down and Oracle standalone.

Other kinds of relationships that can be pushed down

Since, one-to-many relationship guarantees that all related entities pertaining to one fixed entity will end up in the same data node if the data is hash distributed by that fixed entity, it should be obvious that one-to-one relationships between entities also guarantees the same result. Therefore, one-to-one relationships also ensure correctness in data distribution. Whereas, many-to-many relationships should be strictly avoided for deciding on data distribution, as it might lead to incorrect results. For example, people who are aged 25 can live in many counties and many counties can have people who are aged 25 as its residents. Therefore, distributing data by age is deemed a poor design, and in such a distribution force pushing down provides incorrect results. So, many-to-many relationships should be strictly avoided while devising a data distribution strategy.

6.3 Database internals

Performance engineering of queries in a distributed database is challenging due to the multiple components involved and many phases that a query must go through. Furthermore in this work, it is looked at trying to make use of the DiceX framework to drive highly performant analytics websites where small delays can make system unusable. Hence, a deeper understanding of the internal machinery of query processing will help in locating more opportunities for optimization and move us a step closer toward our goal of real-time response to queries over large scale datasets.

6.3.1 Phases in query processing

A query in a distributed database goes through the following five phases in general:

- Coordinator parsing

  This involves parsing the SQL query and identifying the various clauses present in it. This phase also involves building the execution tree for the query, determining the order of execution of clauses present in the query in order to ensure correctness.

- Distributed planning

  This phase involves generating the overall query plan for which operations of the complex query needs to be performed at the data nodes and which at the coordinator. This is the process illustrated in figure 6.3.

- Distributed execution

This is the phase when all remote queries are submitted to the data nodes for intermediate results. It also involves feeding back intermediate results to the data nodes for further execution in a multi-step query processing.

- Local parsing

  This happens at the data nodes where each stand alone database that is launched at the data nodes does its own parsing of the remote query that it had received.

- Local execution

  This denotes the remote query execution that happens locally at the data nodes.

Among these phases, coordinator parsing, distributed planning and local parsing have negligible cost. Execution time both local and at the coordinator is a function of the query plan generated. The approach for optimizing local execution is well understood and comparatively straight forward. Adding indexes to the columns that most often appear on clauses and maximizing hardware resources generally lead to good performance with respect to local execution time.

In contrast, generating a good distributed plan is still not well understood and more study is required. In the following sections, the mechanism for distributed query planning and execution is presented to aid in developing better and possibly more efficient domain specific query plans which in turn can provide faster execution times.

6.3.2 Distributed query planning

Figure 6.3 shows the basics schematics of the distributed query planning that is performed at the postgres-XC coordinator. When the client sends a request with a SQL statement, it is received by the coordinator which performs the query planning. This involves checking whether the given SQL statement is supported by postgres-XC and identifying the `ExecNodes` on which the resultant data is present. Then the requested query is partitioned or simplified if required, sent for execution at the specific data node and the results collected at the coordinator. These results are either given directly to the client or further processed at the coordinator to achieve the desired result. If the SQL statement isn't supported, the query execution is stalled and returned as an error.

When the planner determines that the query is supported, it proceeds with planning and creates the remote query to be executed on the data nodes. These remote queries are solely dependent on the data that is present at the data nodes. When the planner decides that the whole query can be shipped down, it saves time from any further planning and pushes the whole query to the `ExecNodes` identified. If the whole query can't be pushed down, further planning of specific SQL operations and expressions are done. This process is illustrated in figure 6.3. The query planner tries to push down the greatest possible number of operations to

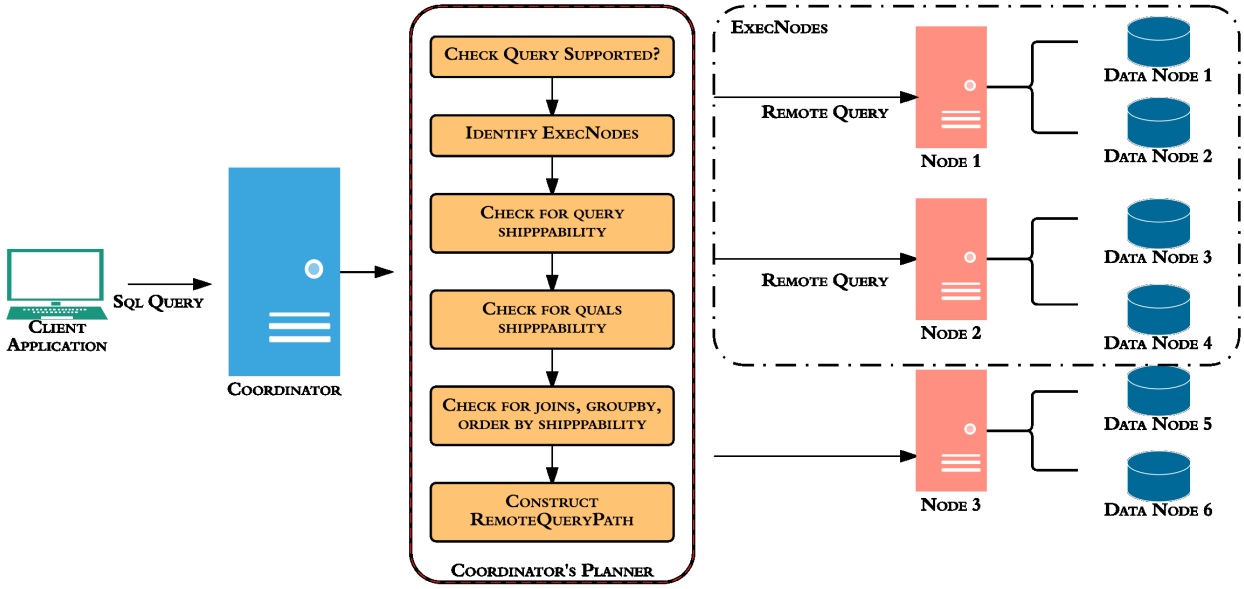


Figure 6.3: Postgres-XC planning phase

the `ExecNodes` in order to achieve maximum parallelism. Planning of specific SQL constructs and the clause shippability criteria have been explained previously in section 6.2.

6.3.3 Distributed execution

The distributed query execution phase as shown in figure 6.4, is further divided into the following steps,

- Submit `RemoteQuery` for local execution at `ExecNodes`

  The remote query received at the ExecNodes are executed in parallel on their local data.

- Receive query results from data nodes at the coordinator

  The coordinator performs I/O multiplexing to read the query results from the `ExecNodes` i.e. it does an asynchronous reading of data on the connection handles to the `ExecNodes`. The unread results from the `ExecNodes` are buffered into their respective connection buffer and read before requesting more data from the particular `ExecNode`. From the connection buffer, the raw bytes are read in data row format and written to a `TupleSlot` object. The coordinator maintains a `TupleStore` for storing the partial

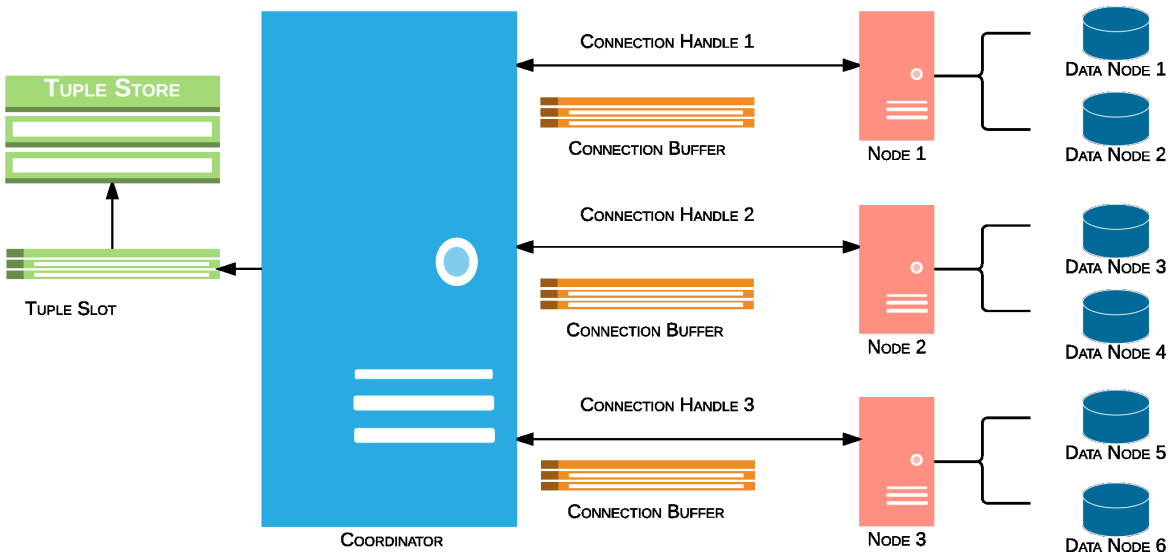


Figure 6.4: Postgres-XC execution phase

results from the data nodes and also for executing any other group aggregations or sort operations that needs to be run at the coordinator as part of the query plan.

- Query execution at the coordinator

Once, the `TupleStore` is filled with all the intermediate results, final query processing is performed at the coordinator if required. If no further processing is required, the gathered query results are simply sent out to the client. For example, consider the following query:

```
SELECT
        SUM(col1)
FROM
        Table1;
```

Say, our table `Table1` is distributed among three data nodes in the cluster. The coordinator takes care of identifying these data nodes in the cluster and sends the query for execution at them to get their local results. The local sum of each `ExecNode` is stored in the coordinator's `TupleStore`. The coordinator then, will run a final `sum` operation with the tuples that are present in its `TupleStore`.

Timing internal functions

When the distributed execution of postgres-XC is studied in detail, it is imperative to analyze the running time of each internal module in the system that is part of this distributed query execution. This kind of analysis can help in making any future improvements to the internal processing engine by identifying and removing any possible bottlenecks.

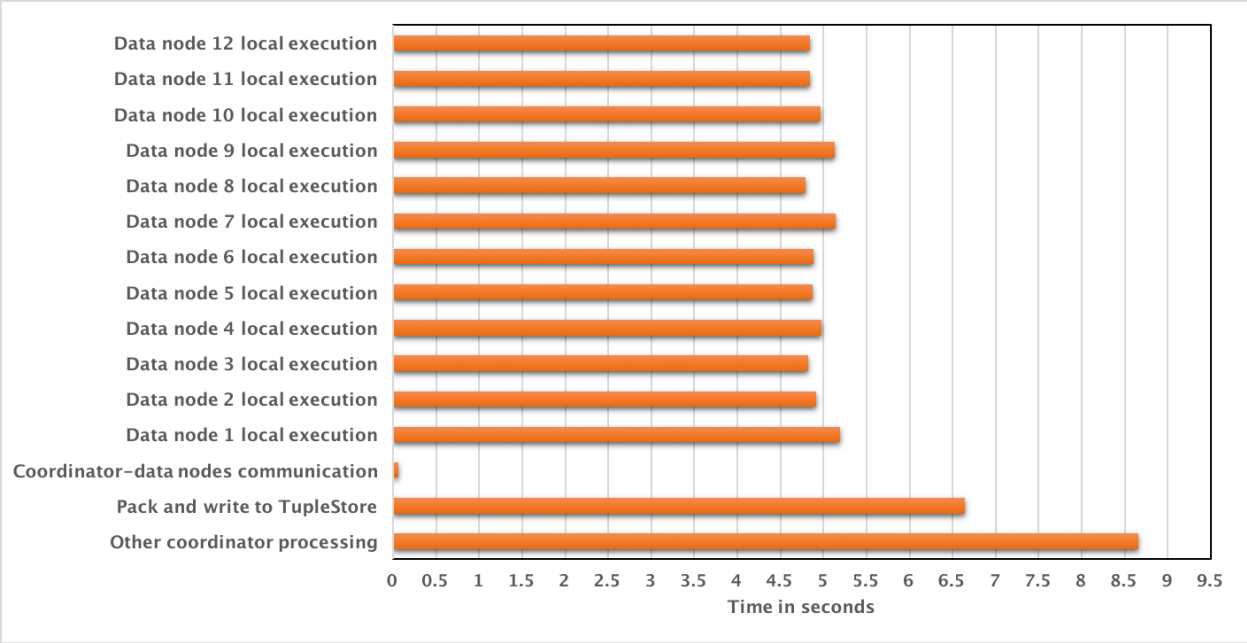


Figure 6.5: Postgres-XC distributed execution of query q9 - internal functions analysis

Major internal functionalities associated with the distributed execution of query q9 from Appendix B are timed separately and presented in the figure. This experiment was conducted with 12 data nodes and with force push down turned off in order to define a clear distinction between the time consumption of each major internal function involved in successful query execution. The total running time of the query was 19.3s.

The following points list the results of our analysis:

- It has to be noted that the I/O multiplexing adopted by the coordinator in receiving results from the data nodes is reducing the communication time normally deemed the bottleneck in such distributed frameworks because they typically use TCP based communication.
- The parallel execution of `Remote Query` at the data nodes, i.e. local execution phase, takes an average of 5s which seems to be normal considering the size of the data involved.

- The local data processing at the coordinator, i.e. packaging the raw data in the form of bytes from the data nodes into data row format and writing to the `TupleStore`, contributes to 30% of the total query execution time. This processing happens in a serial fashion and as per our analysis this is something one should look at first in order to further improve performance. Again, this experiment is done with force push down turned off and so it may not be a problem for queries that can be completely pushed down.

- Query parsing, planning, and other bookkeeping operations in getting the results out to the requesting client are combined together and denoted as 'Other coordinator processing' in the figure.

Chapter 7

Postgres-XL

Postgres-XL(eXtensible Lattice) is a horizontally scalable database cluster which is flexible enough to handle varying database workloads. Horizontal scaling allows scalability by adding more resources to the resource pool and partitioning the data among the resources. Though postgres-XL borrows code base from postgres-XC, it has its own philosophy. It is aimed at providing more stability and enhanced performance with less emphasis on adding new functionality.

7.1 Architecture

The architecture of postgres-XL is almost identical to that of postgres-XC and has the same three major components,

- Global Transaction Monitor(GTM)
- Coordinator
- Data Node

Figure 7.1 shows the simple architecture of postgres-XL showcasing the basic components.

7.2 Postgres-XC vs Postgres-XL

The major difference between postgres-XC and postgres-XL is the way in which certain queries are handled, and the inter data node communication. Postgres-XC does not support communication among the data nodes and prevents them from talking to each other. This

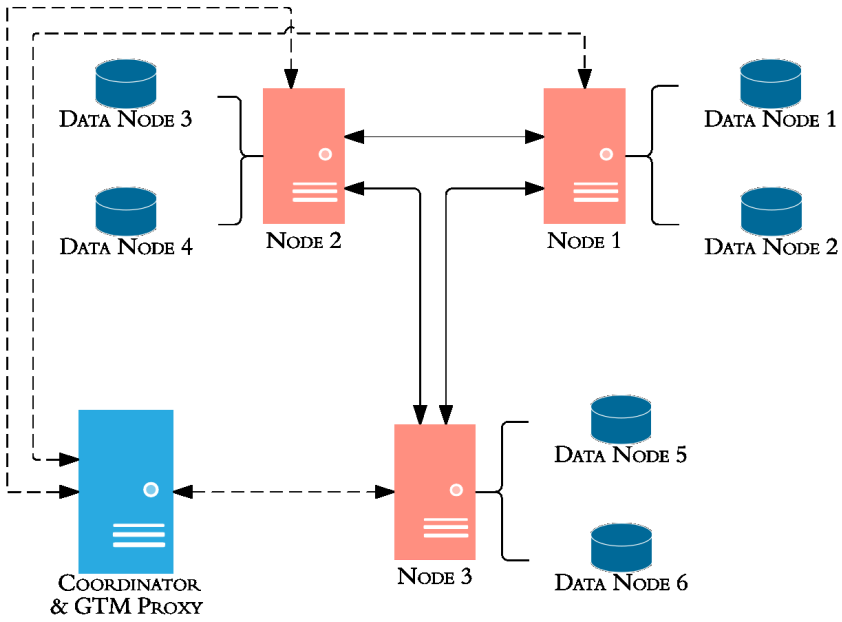


Figure 7.1: Simple Postgres-XL Architecture

will prohibit some join operations to be pushed down to the data nodes and incur more overhead at the coordinator. For example, when there is a requirement for a join between two tables that are in two different data nodes, in postgres-XC, the join operation happens at the coordinator and the data nodes send in all their required table data to the coordinator. This is expensive as nothing is parallelized here and indeed might run slower than a standalone postgreSQL instance. In postgres-XL, every data node knows where the data is stored and communicates with one another and so when a query comes in, it is parsed and planned once on the coordinator and then serialized and sent down to all other data nodes.

7.3 Evaluating DiceX running on postgres-XL

7.3.1 Performance

The DiceX framework can be customized to use either postgres-XC or postgres-XL as the backing distributed database engine. The same set of queries used to evaluate DiceX with postgres-XC were run on DiceX with postgres-XL to make a comparison analysis. Figure 7.2 shows the comparison between the two setups using 4 data nodes each running on its own supercomputing node. The data was hash distributed among the data nodes on the join column. It can be seen that postgres-XC outperforms postgres-XL in certain queries used

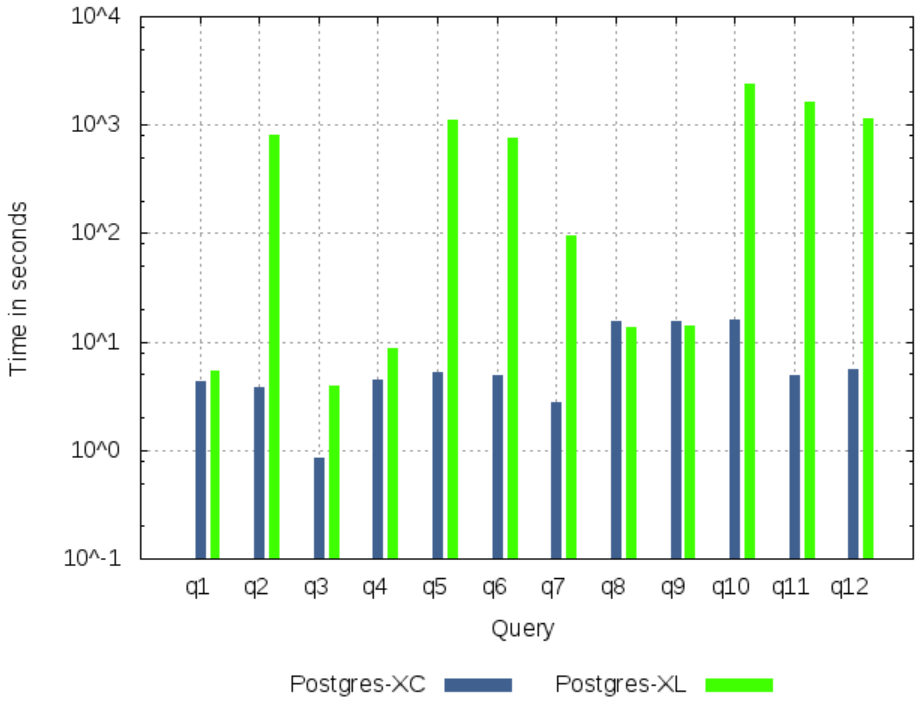


Figure 7.2: Performance comparison of postgres-XL and postgres-XC with 4 data nodes each running on its own compute node for queries from Appendix B

for evaluation and it is observed that such a difference has to be attributed to the inter data node communication and sharing of intermediate results. Further analysis may be required in the future.

7.3.2 Force push down

Forcing push down is an optimization idea devised on a deeper understanding of the domain semantics of the data involved. We haven't implemented force push down in postgres-XL but, the code base has been studied and it is certainly possible to implement it. Furthermore, if the queries are fully pushed down, the behavior of postgres-XL and postgres-XC will become similar as the need for inter data node communication is eliminated.

Chapter 8

Spark SQL

Apache Spark is an open-source cluster computing framework developed at UC Berkeley. Spark's in-memory primitives provide performance up to 100 times faster for certain applications [52]. Spark SQL is a library provided by Spark that brings native SQL support.

8.1 Background

In this section, let us review Apache Spark and Spark SQL's architecture to develop a better understanding of the framework.

8.1.1 Apache Spark

MapReduce framework and its variants have been developed to run data-intensive applications on commodity clusters. But these systems are built around an acyclic data flow model that is not suitable for other popular applications like iterative machine learning algorithms, which apply a function repeatedly on the same dataset and every time data must be reloaded from the disk [55]. Apache Spark was designed to overcome this limitation without compromising the scalability and fault tolerance that MapReduce systems provide. Spark introduces the concept of Resilient Distributed Datasets (RDDs) which is a read-only collection of objects partitioned across a set of machines in the cluster, which can be rebuilt if a partition is lost. Spark can be seen as an alternative to MapReduce but not a replacement. The main intention is to provide a comprehensive and unified solution to manage different big data use cases and requirements.

Key features of Spark includes:

- Less expensive shuffles in data processing

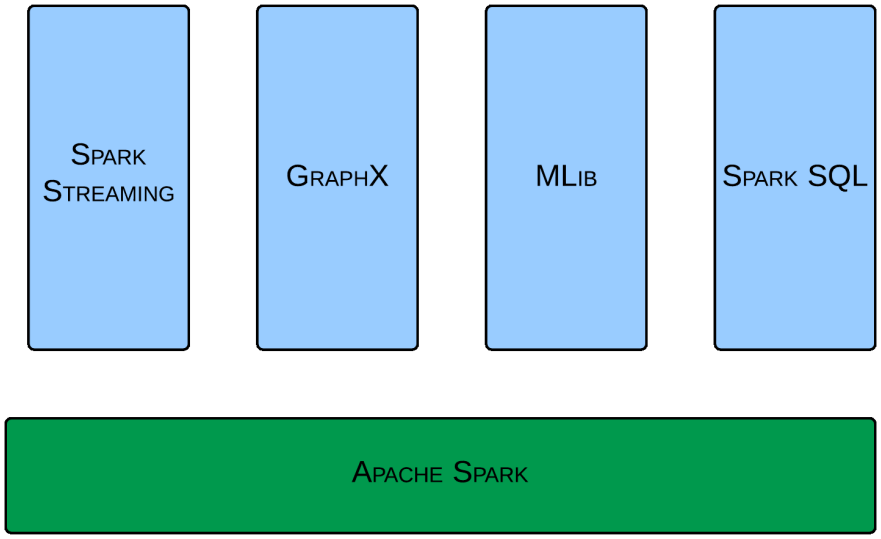


Figure 8.1: Apache Spark Libraries

- Faster performance due to in-memory data storage and near real-time processing
- Holds intermediate results in memory rather than writing them to disk
- Lazy evaluation of big data queries that helps optimizes overall data processing workflow
- Better API support in Scala, Java and Python

Spark Libraries

Apart from the core Spark API, it also supports additional libraries catering to specific applications. Figure 8.1 shows the various libraries that are currently supported by the Spark ecosystem.

- Spark Streaming - Used for processing real-time streaming data
- GraphX - Runs massive graph-parallel computation
- MLib - Scalable machine learning library consisting of common learning algorithms and other utilities
- Spark SQL - Run SQL like queries on Spark data

Spark Architecture

The architecture of Apache Spark is mainly comprised of the following three components:

- API
  Spark API provides developers with the ability to create Spark based applications using a standard API interface. As stated before, Spark provides API support in three languages including Scala, Java, and Python.
- Data storage
  Spark uses Hadoop Distributed File System(HDFS) for data storage. It works with any Hadoop compatible data source including HBase, Cassandra, etc.
- Management framework
  Spark can be deployed as a standalone server or on a distributed computing framework like Mesos or YARN.

8.1.2 Spark SQL

Spark SQL [14] integrates relational data processing with Spark's functional programming API. It offers methods to run native SQL statements to query data present in the Resilient Distributed Datasets (RDD) as well as from other external sources including relational databases. This helps in mixing SQL statements to query external data while performing complex analytics in Spark. Spark SQL allows:

- Importing relational data from Hive tables and parquet files
- Exporting RDDs to Hive tables or parquet files
- Running SQL queries on existing RDDs or imported data

Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two [14]. Spark SQL bridges the gap between relational and procedural data models by providing a `DataFrame API` that can perform relational operations. It must be noted that this `DataFrame API` is similar to the data frame concept available in R but it can perform relational optimizations. Spark SQL also provides a novel extensible optimizer called `Catalyst`.

Spark SQL Architecture

Figure 8.2 illustrates the basic architecture of Spark SQL. It runs on top of Apache Spark as a library and provides SQL interfacing which can be accessed through the command line or

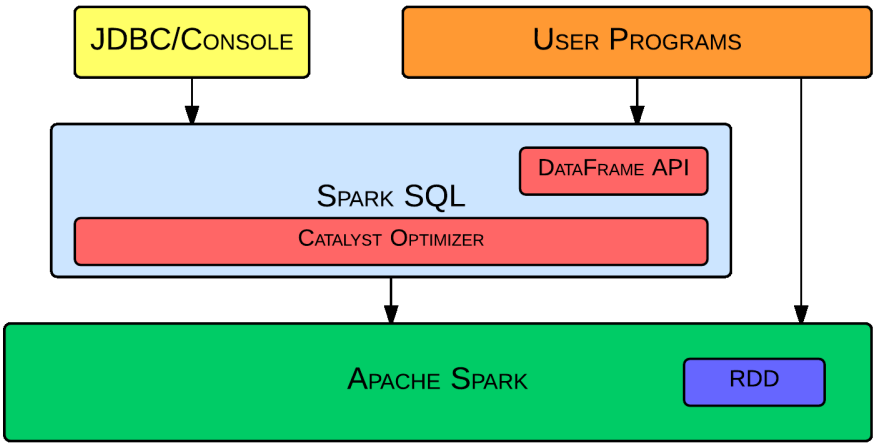


Figure 8.2: Spark SQL Architecture

ODBC/JDBC. User programs written in any of the Spark supported languages can call the Spark SQL interfaces.

DataFrame API

A DataFrame is analogous to a table in a relational database consisting of a collection of rows with a homogenous schema. It can also be manipulated similar to the native distributed collections in Spark i.e. RDDs. These DataFrames can be manipulated with various relational operators such as `where`, `select` and `groupby`[14].

Catalyst Optimizer

Catalyst optimizer is a primitive optimization engine built into Spark SQL. It is designed to be extensible and the creators of Spark SQL expect the community to add more optimizations in the future. The current catalyst optimizer supports both cost-based and rule-based optimizations. Since, functional languages were designed in part to build compilers, Scala was used to build this optimizer[14]. Catalyst contains a general library for representing `trees` and applying `rules` to manipulate them.

8.2 Experiments

Experiments were done to evaluate the performance of Spark SQL to compare with our DiceX framework built on postgres-XC. It seems that Spark SQL performs poorly when compared to postgres-XC. The following sections elucidate the experimental results and findings.

8.2.1 Setup

Spark version 1.3 was launched in a standalone cluster mode with 4 worker nodes and 1 master node on **shadowfax** cluster at **VBI, NDSSL**. Each node has a 12-core Westmere-EP X5670 2.93GHz dual processor with 48 GB of memory. Similarly, the DiceX framework was launched with 4 datanodes and 1 coordinator on the same cluster. The same set of queries given in Appendix B, that were used to evaluate DiceX were run in Spark SQL. These queries are a combination of spatio-temporal queries used in an in-house application called FluCaster, an interactive web application for flu prediction. Data has been loaded from text files and this loading is timed separately and not included in the run time of the query. The schema of these tables are given in Appendix A. The text files containing data are exported from a relational database that FluCaster uses as its back-end. Size of data used for this experiment is given in table 8.1. These queries do a `join` on the `pid` and a `group by` and `order by` on these tables apart from adding some filters on `age`, `gender` and `exposed_time`.

Data Table	Number of Rows
TX_DEMOGRAPHY_INFO	22,000,483
SES_TX_20150416	10,173,311

Table 8.1: Size of data used for the Spark SQL experiment

8.2.2 Query Planning

Before presenting the results of the experiment, it is important to look at Spark SQL's query planning and see how it differs from postgres-XC. Spark SQL query planning involves generating the following:

- Parsed logical plan
- Analyzed logical plan
- Optimized logical plan
- Final physical plan

- Code generation

Consider the query *q5* from Appendix B which involves a `join`, `group by`, `order by` along with some filters.

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        d.age BETWEEN 5 AND 18
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

Spark SQL's final physical query plan for the above query is the following:

```
== Physical Plan ==
Sort [exposed_time#9L ASC], true
 Exchange (RangePartitioning [exposed_time#9L ASC], 200)
  Aggregate false, [exposed_time#9L], [exposed_time#9L,
  Coalesce(SUM(PartialCount#22L),0) AS c1#20L]
   Exchange (HashPartitioning [exposed_time#9L], 200)
    Aggregate true, [exposed_time#9L], [exposed_time#9L,
    COUNT((CAST(pid#5, DoubleType) / 10.0)) AS PartialCount#22L]
     Project [exposed_time#9L,pid#5]
      ShuffledHashJoin [pid#5], [pid#11], BuildRight
       Exchange (HashPartitioning [pid#5], 200)
        Project [pid#5]
         Filter ((age#0L >= 5) && (age#0L <= 18))
          PhysicalRDD [age#0L,blockgroupid#1,county#2,exposed_time#3L,
          gender#4L,pid#5,rep#6L,week#7L,zipcode#8], MapPartitionsRDD[12] at
          mapPartitions at SQLContext.scala:1163
       Exchange (HashPartitioning [pid#11], 200)
        Project [exposed_time#9L,pid#11]
         PhysicalRDD [exposed_time#9L,infectious_time#10L,pid#11,
         recovered_time#12L,rep#13L], MapPartitionsRDD[21] at mapPartitions at
         SQLContext.scala:1163
```

Spark SQL initially does a project on the required columns expected in the output of the query and the join column. This join column is used to hash partition the table rows and

data is shuffled among the worker nodes by means of the `Exchange` method. To reduce the number of rows being shuffled, any filter that is required by the query is applied before the projection. After this, a shuffled hash join is performed on the `pid` column and the result of this join operation is projected. A shuffled hash join involves performing an inner hash join of the two relations by first shuffling the data using the join keys. Finally, to perform the aggregation to count on `pid` and the `group by` on `exposed_time`, the intermediate results are again shuffled among the worker nodes, once for each operation.

The list of operations that are carried out in Spark SQL query processing are the following:

1. Project required columns in the output and the `join` by column from both the tables
2. Shuffle projected data among the worker nodes
3. Perform shuffled hash join
4. Perform aggregation on the `group by` column along with computing local counts on the `pid` column
5. Perform shuffling to exchange results from previous step
6. Do final shuffling of data to compute total counts from all the partial counts
7. Sort by the column `exposed_time`

From the above list, it can be seen that Spark SQL's query plan involves considerable shuffling and data exchanges among the worker nodes. When the number of Spark tasks is increased, this can result in more overhead and lead to a detrimental effect on the query performance.

Postgres-XC query plan for the same query *q5* is the following:

```
GroupAggregate  (cost=49.83..59.84 rows=1 width=64)
   Output: a.exposed_time, count((count((a.pid / 10::numeric))))
   ->  Sort  (cost=49.83..52.33 rows=1000 width=64)
         Output: a.exposed_time, (count((a.pid / 10::numeric)))
         Sort Key: a.exposed_time
         ->  Data Node Scan on "__REMOTE_GROUP_QUERY__"  (cost=0.00..0.00
         rows=1000 width=64)
               Output: a.exposed_time, (count((a.pid / 10::numeric)))
               Node/s: datanode_c2_d1, datanode_c3_d1, datanode_c4_d1,
               datanode_c5_d1
               Remote query: SELECT r.a_1, count((r.a_2 / 10::numeric)) FROM
               ((SELECT d.pid FROM ONLY socialeyes.vt_demography_info d WHERE
               ((d.age >= 5::numeric) AND (d.age <= 18::numeric))) l(a_1) JOIN
               (SELECT a.exposed_time, a.pid FROM ONLY
```

```
socialeyes.ses_vt_20150416 a WHERE true) r(a_1, a_2) ON (true))
WHERE (l.a_1 = r.a_2) GROUP BY 1 ORDER BY 1
```

Looking at the postgres-XC query plan in contrast, almost all the operations involved in the query are pushed down to the data nodes. It can be seen from the remote query that is executed at the data nodes. The intermediate results from data nodes are collected at the coordinator and a final sort operation on the `exposed_time` column and final summing of partial counts are performed at the coordinator.

Postgres-XC's query plan does not involve a significant amount of data exchanges and partitioning as data is already pre-partitioned on the join column and even if it isn't, forcing push down will help in getting a similar query plan even if the join operation is not performed on the distribution column. It has also been analyzed and confirmed that the communication between data nodes and coordinator is asynchronous and therefore does not significantly hinder the query performance.

8.2.3 Results

The time required by Spark SQL to run the queries are shown in figure 8.3 along with the time taken by postgres-XC and postgres-XL. It should be noted that only some of the queries completed successfully and the ones not present in the figure timed out reaching the maximum allocated time of 60 minutes for an individual query execution.

Figure 8.3 shows a comprehensive comparison between postgres-XC, postgres-XL and Spark SQL for the same number of resources, i.e. 4 data nodes in the case of postgres-XC and postgres-XL and 4 worker nodes in the case of Spark SQL.

Spark SQL is still in its primitive stages and lacks serious optimizations in query performance in order for it to be used as a backing data management system for an on-line or web application. It is evident from the results in figure 8.3. This may be attributed to the lack of a better query optimizer in Spark SQL. The Catalyst optimizer is still in its early stages and would require more optimizations in future. When a join operation is performed with at least one equality relationship between two tables, Spark SQL will automatically hash partition the data before performing the join but, it still doesn't allow pre-partitioning of data on a specific column that would enable it to avoid extra shuffling and help in propagating this pre-partitioning information from the in-memory cached representation. This is a work in progress at the Spark SQL developers community and is anticipated to be available with the future releases. In the current scenario, like any MapReduce based frameworks, Spark SQL cannot also be used as a reliable high performance data source for highly responsive web applications. A new investigation of Spark SQL is recommended when the above mentioned pre-partitioning feature is added to it in the future.

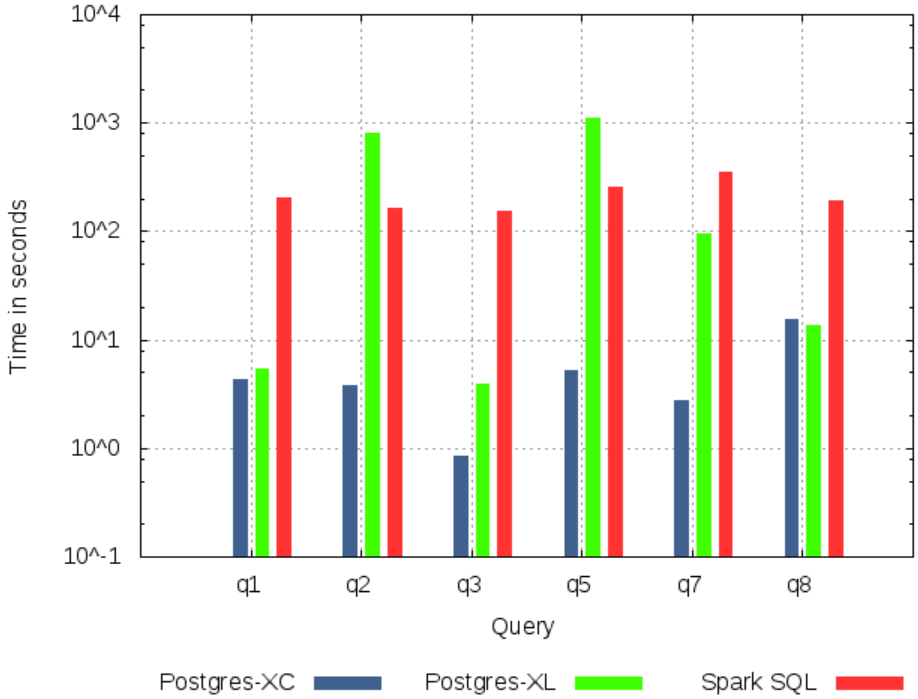


Figure 8.3: Comparison of time taken by Postgres-XL vs Postgres-XC vs Spark SQL for queries from Appendix B for 4 data nodes or 4 worker nodes

Can the idea of force push down apply to Spark SQL?

Force push down is the idea discussed in chapter 6 with respect to postgres-XC where we applied a domain semantics based data distribution and forcefully pushed down all time consuming SQL operations to the data nodes to achieve maximized parallel performance. If one takes a closer look at the Spark SQL's query plan, it is easy to see that it already performs major time consuming SQL operations like `joins` in parallel. So, in a way, it is already pushing major operations to the worker nodes in order to execute them in parallel. At the same time, it does not do all the operations together at the same time i.e. it handles the projection of required columns in parallel initially followed by the shuffled hash join in parallel. If these multiple steps can be integrated and done at the same time, then the quantity of data exchanges and shuffling can be reduced. By comparison, postgres-XC pipelines operations, so that all can be executed at the same time without shuffling of intermediate results. In spark SQL, this kind of pipelining is not built in and if we have to enforce such a behavior, we predict that it involves serious changes to its query planner.

Is communication the real bottleneck?

We have seen that Spark SQL performs poorly when compared to postgres-XC and from the query planner; it has been found that Spark SQL does a lot of shuffling of intermediate results to arrive at the final results. We suspect that these shufflings and the associated communication, which is actually TCP based, could be the major reason for its poor performance. In order to prove that the communication between the worker nodes and master node is the major bottleneck, a deeper inspection of the internal communication module is required and highlighted as a future study.

Chapter 9

DiceX In Use

Previous evaluations of the DiceX framework were performed by running some spatio-temporal queries integral to a web application SocialEyes designed and developed at **NDSSL, VBI**. It should be understood that DiceX can be extensively used for any kind of application that requires high performance relational query processing and is not just limited to running offline evaluations. Connecting DiceX framework with an online application and using it as a data management system for that application will be a good example to show the multi-faceted possibilities of the framework. FluCaster [2], an application that is an extension of SocialEyes is an easy choice of interest for this purpose. Like SocialEyes, FluCaster was also designed and developed at **NDSSL, VBI**.

9.1 FluCaster

FluCaster is a pervasive web application that provides a spatio temporal visualization of influenza attack in the past and forecast for the future. Such an application demands a high performance database engine for querying huge amounts of data to visually present the disease infection rates. It is assumed that use of a distributed database framework like DiceX will definitely enrich the user experience.

9.1.1 Architecture

Figure 9.1 shows the current architecture of FluCaster deployed using an Oracle database. EpiFast [17] is a novel algorithm for large scale realistic epidemic simulations on distributed memory systems. It is based on a unique interpretation of stochastic disease propagation in a contact network. EpiFast is used as the simulation engine for producing the flu predictions. The results of these predictions are loaded into an Oracle relational database hosted on

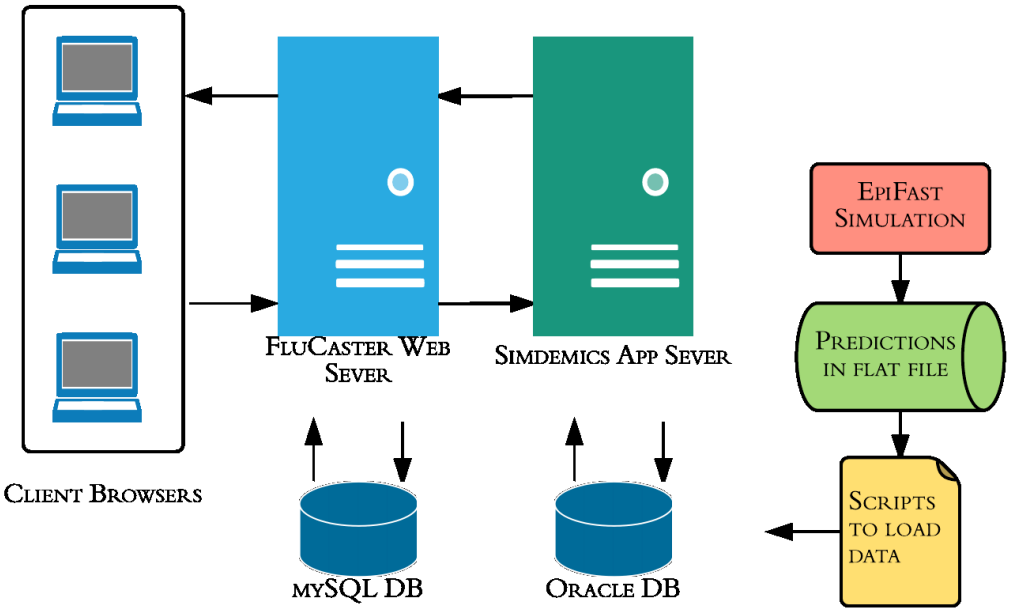


Figure 9.1: FluCaster Architecture

its own server. FluCaster is deployed in a three tier architecture as shown in the figure. Simdemics is the API engine that accesses the database and exposes RESTful APIs to provide the infection data in a JSON [5] format as required by the user interface of the web application. The web server acts as the first-hand interface to the client browsers accessing the application.

9.2 FluCaster on DiceX

The motivation is to replace the Oracle database from the existing FluCaster setup with the DiceX framework and make it feed infection data to the Simdemics APIs. The Simdemics application server is hosted on **socialeyes** VM at **VBI, NDSSL** running a Intel(R) Xeon(R) processor with 4GB memory. DiceX framework was launched with 12 data nodes (2 per compute node) and 1 coordinator on **shadowfax** cluster at **VBI, NDSSL**. Each of these nodes run a 12-core Westmere EP X5670 2.93GHz dual processor with 48GB of memory. The relational data required by FluCaster is loaded onto a newly launched database engine from the Oracle database using the parallel data ingestion utility supported by DiceX.

9.2.1 SSH Tunneling

Simdemics uses JDBC connectivity protocol to communicate with the database and configurational changes are made to use the JDBC driver for PostgreSQL instead of Oracle. Since, the distributed database engine runs behind the firewall, SSH tunneling via port forwarding is used to connect the Simdemics app server to the database. A dedicated port has been opened at the login nodes of the **shadowfax** cluster for access from the **socialeyes** virtual machine. A SSH tunnel is created between the login node and the compute node that runs the coordinator and all requests to the port opened at the login node were forwarded to the port in which the database engine postgres-XC's coordinator process is running on its dedicated compute node. This is a guaranteed secure communication method and has been proven successful.

9.2.2 Evaluation

It has been shown in chapter 6 that the force pushing down of time consuming operations like `join` and `group by` reduces the running time of queries. Since FluCaster is a direct extension of SocialEyes, almost all queries it issues are identical to SocialEyes and therefore it is observed that using DiceX with force pushing down turned on, enriches the user experience of FluCaster in terms of speed and responsiveness. The results obtained in figure 6.2 shows the difference in time costs with and without force pushing down. Even with force pushing down turned off, DiceX would easily outperform Oracle. This has been confirmed in previous research [37]. Therefore, it is not included as part of this discussion.

Some Limitations

- With the limited set of queries that SocialEyes and FluCaster uses, it cannot be yet proven that the DiceX framework can possibly reduce the running time of complex time consuming operations on macroscopic data. Complex queries that involve various kinds of joins other than equijoin or inner join with a combination of group bys are yet to be studied.
- Some commonly used mutable functions in SQL queries can introduce a huge bottleneck in query performance as they are not pushed down to the data nodes. More sophisticated query planning needs to be introduced in the planner module to handle the execution of these mutable functions without compromising performance. It is expected that these mutable functions are more commonly used when querying spatio-temporal data.
- Though query processing time has been drastically reduced, some Simdemics APIs are still deemed slow because of the bottlenecks in the JSON data packaging methodology adopted in the application code. If this can be rectified in the application code,

the complete effect of using DiceX will be visible to the users of FluCaster. The JDBC/ODBC free API design to use DiceX services as described in chapter 10 is an attempt to overcome this bottleneck.

Chapter 10

Query Distribution and Execution Broker

Query distribution and execution broker is a web service aimed at providing a liaison between the DiceX distributed database framework and any application requesting its service. This broker exposes RESTful APIs that any application can make use of irrespective of the programming environment.

10.1 Motivation

There are several motivations behind the design and implementation of this query distribution and execution broker. Some of them are:

- Enable usage of the DiceX framework by any application irrespective of the programming language or framework used to build the application and provide a database server interface that receives query requests for specific application. Execute the query and provide output in a JavaScript Object Notation(JSON) format which any language can interpret.
- Provide an efficient DiceX cluster management system to keep track of the currently available distributed database engines running on the supercomputer and servicing specific applications.
- Assist in providing a DNS like lookup service for the applications to lookup from the list of databases that are registered with the broker and start launching queries to the one that is pertaining to the application.
- In chapter 9, it was elucidated how DiceX can be used as a backing data management system for a high performance web application like FluCaster. SSH tunneling is used to

bypass the firewall and connect to the DiceX distributed database running on **shadowfax** supercomputing cluster. Though SSH tunneling has proven to be working well, it has its own limitations. When a particular port is bound to a SSH tunnel, then no other connections can be made to the same port. So, when multiple applications are required to use their own distributed database, it is necessary to open individual ports and create SSH tunnels for each of these applications. This is undesirable and a more pragmatic solution is needed. With the broker, the need to create a SSH tunnel every time a new application seeks access to its database is eliminated.

- To enable a JDBC/ODBC free application development experience.

10.2 Design

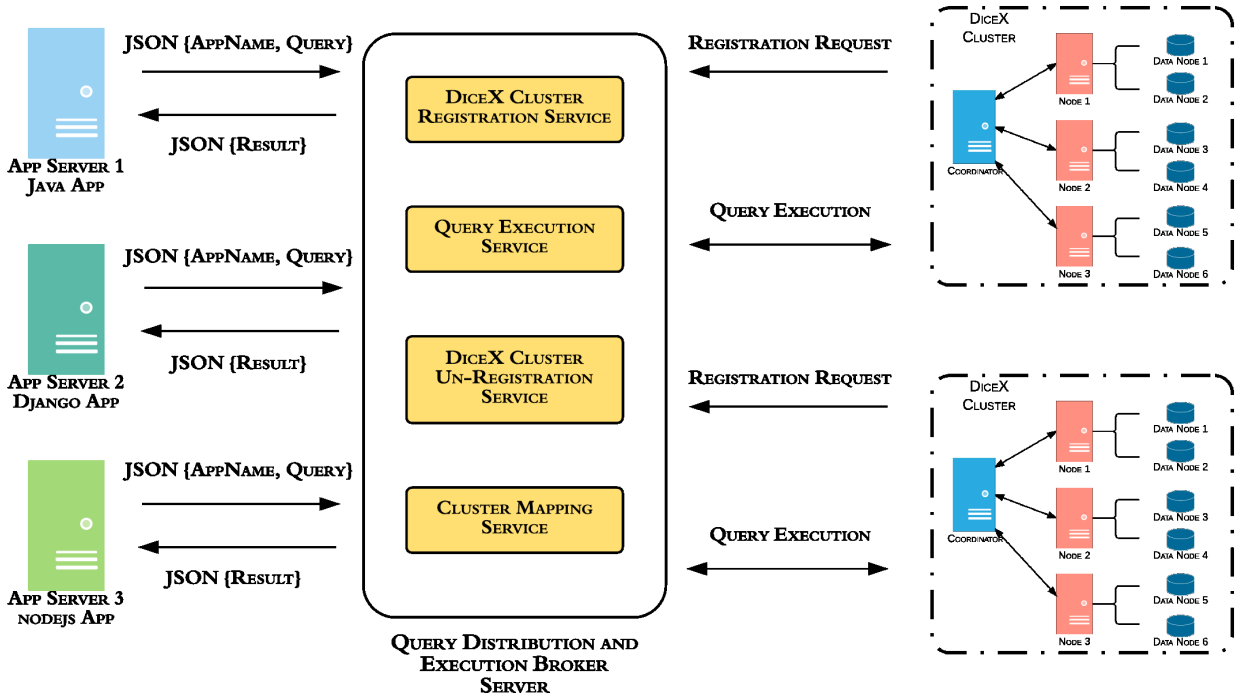


Figure 10.1: Query distribution and execution broker architecture

The basic schematic design of the query distribution and execution broker is shown in figure 10.1. The broker exposes RESTful APIs that both the DiceX framework and applications invoke through HTTP GET and POST requests. When a new distributed database is launched, it issues a registration request to the broker. The broker, in turn, creates a new `Coordinator`

object and stores the relevant information about the postgres-XC coordinator, including the host name, port number, database name, etc. Irrespective of the programming language used to develop them, applications will send a HTTP request with the query and database name packed in a JSON format. The broker will handle matching to the appropriate database and will execute the query provided. The results of the query are packed into JSON and are sent back as an HTTP response to the application.

This design addresses all the motivations listed and is elegant and simple. Since the broker is implemented as a web service that exposes RESTful APIs, it is easily extensible and more API endpoints can be added in the future as required without worrying about scalability. We claim that the broker service is generic because all communications to and from the broker are done only through JSON which is a lightweight data-interchange format and completely programming language independent [5].

10.3 Implementation

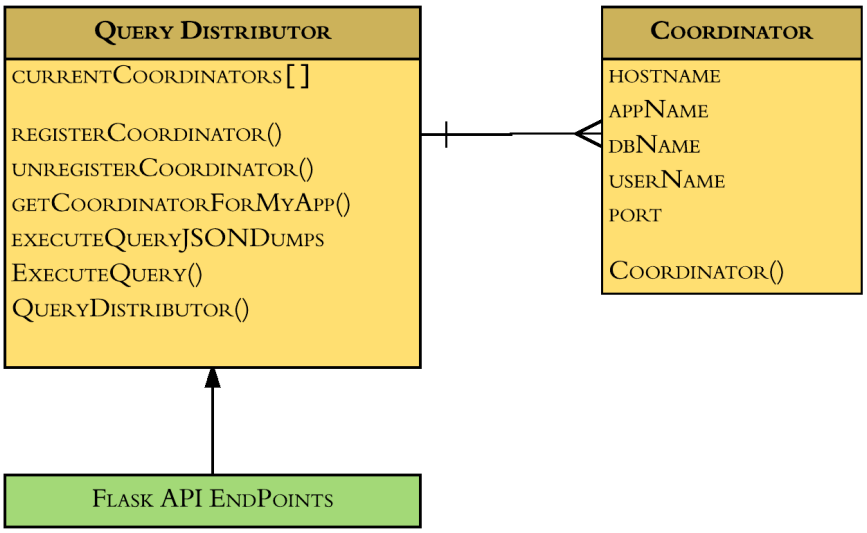


Figure 10.2: Query distribution and execution broker object oriented design

As shown above, the broker can be implemented as a simple web service and has currently been implemented using a REST framework for python called Flask. Figure 10.2 gives the schematics of the object oriented design of the broker. Currently, the broker stores its data in memory and can be extended in the future to use a simple persistent storage like a light weight *sqlite* database. Every time a new distributed database is launched using DiceX framework, a new `Coordinator` object is created with its postgres-XC coordinator

information. This object is then added to the `currentCoordinators` collection maintained at the `Query Distributor`. In this way, a one-many relationship is established between these entities. All the Flask endpoints make use of the methods exposed in the `Query Distributor` entity.

10.3.1 Python Flask

Flask is an open-sourced microframework for python. With Flask, it is easy to create and host RESTful API endpoints. It depends on two external python libraries, namely jinja2 (a templating engine) and the Werkzeug, a WSGI(Web Server Gateway Interface) utility toolkit. A python method can be converted to an API endpoint simply by associating a Flask route function call to it. Usage of Flask in the web development community has increased recently and it is proven to be a successful framework for creating web services [49]. Flask has been chosen for implementing the broker as it is powerful and at the same time easy to develop and maintain.

10.4 Application Programming Interfaces(APIs)

The query distribution and execution broker currently offers the following APIs and it can be extended when new APIs are required in the future.

10.4.1 GET

`/executeQueryJSONDumps`

- Executes the given query on the distributed database pertaining to the given application, if the database is registered to the broker. The raw SQL results are dumped into JSON format at the broker. Since this is a GET request, the parameters are sent as query parameters in the URL.
- Parameters
  - `query` : SQL query to run
  - `appName` : Name of the application seeking the query. This is used to match with the current databases registered to the broker.
- Returns
  Query result in JSON format or an empty JSON object in case of exceptions.

`/executeQuery`

- Executes the given query on the distributed database pertaining to the given application, if the database is registered to the broker. The major difference from the above listed `/executeQueryJSONDumps` API is that the conversion of query result to JSON is performed by the database. PostgreSQL supports generating JSON output of the query result. Since, this is a GET request, the parameters are sent as query parameters in the URL.
- Parameters
  - `query` : SQL query to run
  - `appName` : Name of the application seeking the query. This is used to match with the current databases registered to the broker.
- Returns
  Query result in JSON format or an empty JSON object in case of exceptions.

`/getCurrentCoordinators`

- Parameters
  - None
- Returns
  List of all the distributed databases that are currently registered with the broker. For each database, it provides the application name, corresponding database name, host name of the super computing node where the postgres-XC coordinator runs and the port number. This response is again in JSON format.

10.4.2 POST

`/registerCoordinator`

An important API endpoint invoked by the DiceX launch or resume utility whenever a new distributed database is being launched for use by an application. The launch and resume utilities can be configured whether to register or not depending upon user request.

- Parameters - These input parameters are expected to be sent in JSON format.
  - `hostname`: Host name of the supercomputing node where the postgres-XC coordinator is running

  - `appName` : Name of the application for which the database will store data. This is used to map the incoming query request from the application with the appropriate database by the broker
  - `dbName` : Name of the database under which the schema exist
  - `port` : Port number of the postgres-XC coordinator process
  - `userName` : Owner of the database

- Returns
  A message in JSON format on whether the registration was successful.

`/unregisterCoordinator`

Invoked to cancel registration of a distributed database that has already been registered with the broker.

- Parameters - These input parameters are expected to be sent in JSON format.
  - `appName` : Name of the application whose database needs to be unregistered from the broker
- Returns
  A message in JSON format on whether the cancellation was successful.

`/executeQueryJSONDumpsPost`

Same as the GET API executeQueryJSONDumps listed above but, the input parameters are required to be sent in JSON format instead of the URL.

`/executeQueryPost`

Same as the GET API executeQuery listed above but, the input parameters are required to be sent in JSON format instead of the URL.

`/getCoordinatorForMyApp`

To introspect and find the corresponding distributed database for the input application. This is also internally used by other APIs to find the matching `Coordinator` object.

- Parameters - These input parameters are expected to be sent in JSON format.

  - `appName` : Name of the application whose database information is of interest

- Returns
  Returns the corresponding `Coordinator` object containing information about the distributed database if there is a matching application in the current list of registered applications. If a match is not found, an empty JSON object is returned.

10.5 Scope for new application development strategy

The broker can be offered as a complementary utility of DiceX to effectively connect to it from other applications including webapps. Applications that require the usage of DiceX for its backing data store can be designed in such a way that the database connectivity, query execution, and query result processing be ignored in the application code and instead delegated to the query distribution and execution broker's APIs. Though the broker service is tested by modifying some basic Simdemics APIs, it hasn't been tested with an entire application. More detailed study and analysis is required in the future, by building an application completely devoid of traditional JDBC/ODBC and using only the broker, in order to gain a better understanding of the merits and demerits of such a design. Nevertheless, the broker service guarantees a programming language independent, flexible usage of the services of DiceX along with the other benefits.

Chapter 11

Conclusion and Future Works

Epidemiological analyses are mostly performed over large volumes of data and these data-intensive tasks cannot be efficiently performed by only using traditional databases and custom written programs. This was our basic motivation behind trying to use supercomputing resources to perform these data-intensive tasks in manner similar to that for compute-intensive tasks. Through the DiceX framework and its associated workflows, we have shown that this idea is capable of accomplishing what was intended. By using the postgres-XC distributed database as the core relational data processing engine, we were able to perform time consuming data analyses and query processing typical to computational epidemiology with real-time response rates. Apart from the default high speed parallel query execution supported by postgres-XC, we have utilized our knowledge of domain semantics in devising an efficient data distribution strategy and forced pushing down of the entire SQL query to the data nodes as a means of further improving query processing times. We have shown that forcing query push down has indeed helped in reducing the computation time by 40%-60%, and achieving such a throughput does greatly favor complex analyses over large datasets.

The cloud computing like workflows supported by the DiceX framework does showcase its multipurpose usage for a variety of researches, not limited to the field of computational epidemiology. The freeze & migrate and resume workflow can be easily adopted and used for machine learning algorithms that require running the same algorithm repeatedly with the intermediate results being used for the next iteration. Introducing the concept of cluster elasticity enables the framework to scale dynamically depending upon new needs for more resources as well as the availability of resources at the time of initial requests. Finally, implementing a generic RESTful interface to access the framework through the query distribution and execution broker service allows any application to communicate with the framework without having to worry about programming semantics and intricacies. Thus, the thesis produces an "all-round" improvement to the existing basic DiceX framework [37] and enables a multi-faceted usage for relational computing using HPC resources and supercomputers.

With respect to future extensions, the DiceX framework provides further possibilities and

opportunities to enhance and expand. Following are a few that are worth mentioning:

- The idea of forcing query push down to the data nodes is solely based on the type of relationships that the underlying data engages and it cannot be applied when such essential conditions doesn't satisfy. In those scenarios, we will end up with the default query planning and execution offered by postgres-XC. After analysing the internal query execution and timing essential internal functions, we found that packaging results from the data nodes and writing to the `TupleStore` at the coordinator takes 30% of the total query execution time. There is an opportunity to optimize this piece of internal operation by means of parallelizing this process, say using simple `pthread` library or cilk plus. This will bring about abundant changes to the distributed query execution module of postgres-XC.

- A cost benefit analysis can be made by comparing the DiceX framework with Amazon Web Services(AWS) and other similar frameworks with respect to pricing structure and monetary benefits.

- Running database benchmarks like TPC-H and TPC-DS and comparing with other state of the art systems with respect to performance and scalability.

- The concept of elastic cluster currently focuses only on adding new resources in an ad-hoc fashion. Ad-hoc removal of resources hasn't been implemented or studied.

- Extending the query distribution and execution broker by adding more API endpoints that can take care of the freeze & migrate and resume as well as the ad-hoc data node addition services. The broker service can be made as the primary interface to interact with the entire distributed DiceX framework for all of its services, not limited to query distribution and execution.

- With the current postgres-XC architecture, communication between the data nodes and coordinator is through TCP communication channels and data is received at the coordinator in the form of raw bytes and is packed into data row format. Even though we have identified that communication isn't a bottleneck in query processing because of the usage of I/O multiplexing, a complete redesign of the mode of communication can be explored. Using MPI or Charm++ [1] for communication can help in highly parallelizing data transfer between the data nodes and coordinator and also between the coordinator and the user process. Moreover, this total revamp of the architecture could also help in removing the bottleneck that we identified in writing to the `TupleStore` because MPI or Charm++ like parallel programming models support parallel I/O and can be used for writing to the `TupleStore`.
  By incorporating parallel programming models for performing parallel I/O, we would be adding an extra communication channel that runs as a separate process above the data node and coordinator processes. This kind of design enables integrating High

Performance Computing closely to the DiceX architecture with respect to communication.

- Implementing force push down on DiceX using postgres-XL as its core relational data processing engine.

Appendix A

Evaluation Schema

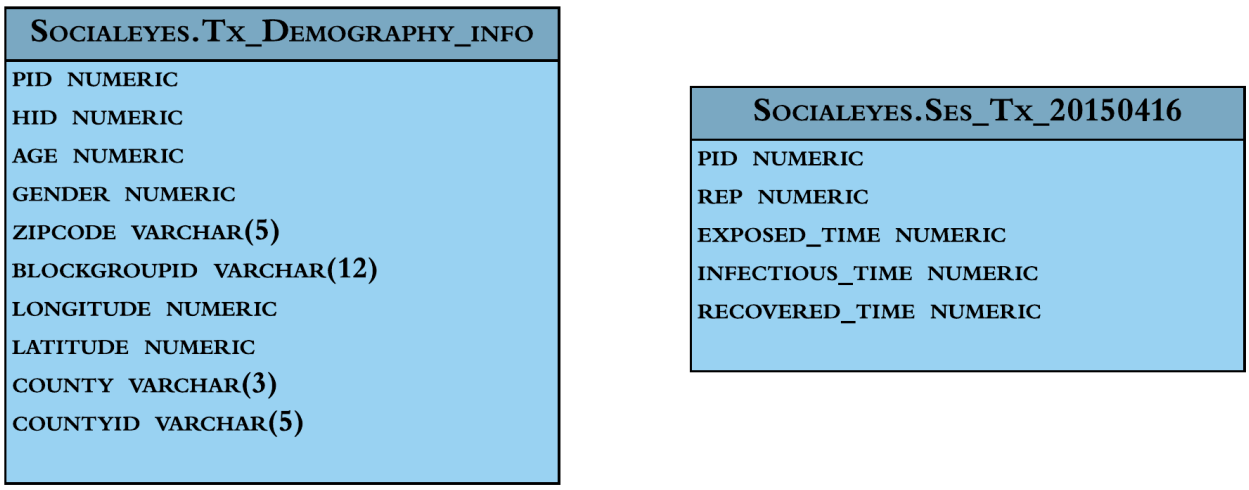


Figure A.1: Schema of the tables used for evaluation

Appendix B

Evaluation Queries

1. *q1*

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.SES_TX_20150416 a
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

2. *q2*

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        d.countyid = '48201'
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

3. *q3*

```
SELECT
```

```
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        d.blockgroupid='482015555001'
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

4. *q4*

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d
WHERE
        exposed_time BETWEEN
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000','DD Mon YYYY')+1-365) AND
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000', 'DD Mon YYYY'))
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

5. *q5*

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        d.age BETWEEN 5 AND 18
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

6. *q6*

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        (d.age BETWEEN 0 AND 4 OR d.age BETWEEN 5 AND 18) AND
        d.gender=1 AND
        exposed_time BETWEEN
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000','DD Mon YYYY')+1-365) AND
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000', 'DD Mon YYYY'))
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

7. *q7*

```
SELECT
        exposed_time effectedDate, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        d.countyid='48201'AND
        d.age BETWEEN 5 AND 18 AND
        d.gender=1 AND
        exposed_time BETWEEN
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000','DD Mon YYYY')+1-365) AND
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000', 'DD Mon YYYY'))
GROUP BY
        exposed_time
ORDER BY
        exposed_time;
```

8. *q8*

```
SELECT
        d.countyid COUNTYID, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
```

```
        d.pid = i.pid
GROUP BY
        d.countyid
ORDER BY
        d.countyid DESC;
```

9. *q9*

```
SELECT
        d.blockgroupid blockgroupid, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid
GROUP BY
        d.blockgroupid
ORDER BY
        d.blockgroupid DESC;
```

10. *q10*

```
SELECT
        d.countyid COUNTYID, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid AND
        exposed_time BETWEEN
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000','DD Mon YYYY')+1-365) AND
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000', 'DD Mon YYYY'))
GROUP BY
        d.countyid
ORDER BY
        d.countyid DESC;
```

11. *q11*

```
SELECT
        d.blockgroupid blockgroupid, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
```

```
        d.pid = i.pid     AND
        d.age BETWEEN 5 AND 18
GROUP BY
        d.blockgroupid
ORDER BY
        d.blockgroupid DESC;
```

12. *q12*

```
SELECT
        d.blockgroupid blockgroupid, ROUND(COUNT(a.pid)/10) COUNT
FROM
        SOCIALEYES.TX_DEMOGRAPHY_INFO d, SOCIALEYES.SES_TX_20150416 a
WHERE
        d.pid = i.pid     AND
        (d.age BETWEEN 0 AND 4 OR d.age BETWEEN 5 AND 18) AND
        d.gender=1 AND
        exposed_time BETWEEN
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000','DD Mon YYYY')+1-365) AND
        TRUNC(CURRENT_DATE-TO_DATE('01 Jan 2000', 'DD Mon YYYY'))
GROUP BY
        d.blockgroupid
ORDER BY
        d.blockgroupid DESC;
```

Bibliography

[1] Charm++. http://charm.cs.uiuc.edu/research/charm. [Online; accessed 06-August-2015].

[2] FluCaster. http://socialeyes.vbi.vt.edu:8443/flucaster/flucaster.html. [Online; accessed 14-July-2015].

[3] hadoop. https://hadoop.apache.org/. [Online; accessed 11-July-2015].

[4] HIVE. http://hive.apache.org/. [Online; accessed 11-July-2015].

[5] Introducing JSON. http://json.org/. [Online; accessed 13-July-2015].

[6] Postgres-XC. https://wiki.postgresql.org/wiki/Postgres-XC. [Online; accessed 12-July-2015].

[7] Postgres-XC 1.2.1 Documentation. http://postgres-xc.sourceforge.net/docs/1_2_1/sql-createtable.html. [Online; accessed 05-July-2015].

[8] ScaleBase Whitepapers. https://www.scalebase.com/resources/whitepapers/. [Online; accessed 08-July-2015].

[9] Worldometers population. http://www.worldometers.info/world-population/. [Online; accessed 10-July-2015].

[10] Google Flu Trends. https://www.google.org/flutrends/us/#US, 2011. [Online; accessed 01-July-2015].

[11] Postgres-XC Concept, Implementation and Achievements. http://postgres-x2.github.io/presentation_docs/2014-07-PGXC-Implementation/pgxc.pdf, 2014. [Online; accessed 01-July-2015].

[12] Michel E. Adiba and Bruce G. Lindsay. Database snapshots. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 86–91. VLDB Endowment, 1980.

[13] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[14] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark.

[15] Christopher L. Barrett, Keith R. Bisset, Stephen G. Eubank, Xizhou Feng, and Madhav V. Marathe. Episimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 37:1–37:12, Piscataway, NJ, USA, 2008. IEEE Press.

[16] Michael D Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457 – 1478, 2001. Clusters and computational grids for scientific computing.

[17] Keith R. Bisset, Jiangzhuo Chen, Xizhou Feng, V.S. Anil Kumar, and Madhav V. Marathe. Epifast: A fast algorithm for large scale realistic epidemic simulations on distributed memory systems. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 430–439, New York, NY, USA, 2009. ACM.

[18] Keith R. Bisset, Jiangzhuo Chen, Xizhou Feng, Yifei Ma, and Madhav V. Marathe. Indemics: An interactive data intensive framework for high performance epidemic simulation. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 233–242, New York, NY, USA, 2010. ACM.

[19] K.R. Bisset, A.M. Aji, E. Bohm, L.V. Kale, T. Kamal, M.V. Marathe, and Jae-Seung Yeom. Simulating the spread of infectious disease over large realistic social networks using charm++. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 507–518, May 2012.

[20] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, March 1990.

[21] D. Brugali and M.E. Fayad. Distributed computing in robotics and automation. *Robotics and Automation, IEEE Transactions on*, 18(4):409–420, Aug 2002.

[22] Cheng Chen, Zhong Liu, Wei-Hua Lin, Shuangshuang Li, and Kai Wang. Distributed modeling in a mapreduce framework for data-driven traffic flow forecasting. *Intelligent Transportation Systems, IEEE Transactions on*, 14(1):22–33, March 2013.

[23] C.D. Corley and A.R. Mikler. A computational framework to study public health epidemiology. In *Bioinformatics, Systems Biology and Intelligent Computing, 2009. IJCBS '09. International Joint Conference on*, pages 360–363, Aug 2009.

[24] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, April 2013.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[26] Kaushik Donkena and Subbarayudu Gannamani. Performance evaluation of cloud database and traditional database in terms of response time while retrieving the data. *Electrical Engineering*, 2012.

[27] Christos Doulkeridis and Kjetil Norvag. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, June 2014.

[28] Stephen Eubank. Scalable, efficient epidemiological simulation. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 139–145, New York, NY, USA, 2002. ACM.

[29] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for mapreduce queries. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 26–37, New York, NY, USA, 2012. ACM.

[30] G. Galante and L.C.E. de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270, Nov 2012.

[31] Dan Gillick, Arlo Faria, and John DeNero. Mapreduce: Distributed computing for machine learning. *Berkley, Dec*, 18, 2006.

[32] S. M. Shamimul Hasan, Sandeep Gupta, Edward A. Fox, Keith Bisset, and Madhav V. Marathe. Data mapping framework in a digital library with computational epidemiology datasets. In *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '14, pages 449–450, Piscataway, NJ, USA, 2014. IEEE Press.

[33] Miguel A Hernán and David A Savitz. From" big epidemiology" to" colossal epidemiology": when all eggs are in one basket. *Epidemiology*, 24(3):344–345, 2013.

[34] Jian Huang, Shaoqing Qiao, Haitao Yu, Jinhui Qie, and Chunwei Liu. Parallel map matching on massive vehicle gps data using mapreduce. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1498–1503. IEEE, 2013.

[35] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[36] Eric Jain. Distributed computing in bioinformatics. *Applied bioinformatics*, 1(1):13–20, 2002.

[37] Mohammed Saquib Akmal Khan. Efficient Spatio-Temporal Network Analytics in Epidemiological Studies using distributed databases. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2014.

[38] Jianzhong Li, Hong Gao, Jizhou Luo, Shengfei Shi, and Wei Zhang. Infinitedb: A pc-cluster based parallel massive database management system. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 899–909, New York, NY, USA, 2007. ACM.

[39] Kuo-Bin Li. Clustalw-mpi: Clustalw analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003.

[40] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[41] Ming Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430, June 2012.

[42] Madhav Marathe and Anil Kumar S. Vullikanti. Computational epidemiology. *Commun. ACM*, 56(7):88–96, July 2013.

[43] Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, February 1997.

[44] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6(14):1702–1713, September 2013.

[45] Ju-Won Park, Jae Keun Yeom, Jinyong Jo, and Jaegyoon Hahm. Elastic resource provisioning to expand the capacity of cluster in hybrid computing infrastructure. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 780–785, Washington, DC, USA, 2014. IEEE Computer Society.

[46] Slawomir Pilarski and Tiko Kameda. A novel checkpointing scheme for distributed database systems. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 368–378, New York, NY, USA, 1990. ACM.

[47] Xuan Ping. Selection of nodes for distributing relations in parallel database. In *Informatics in Control, Automation and Robotics (CAR), 2010 2nd International Asia Conference on*, volume 1, pages 162–165, March 2010.

[48] Kaveh Razavi, LiviuMihai Razorea, and Thilo Kielmann. Reducing vm startup time and storage costs by vm image content consolidation. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, StephenL. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 75–84. Springer Berlin Heidelberg, 2014.

[49] Armin Ronacher. Powered By Flask. `http://flask.pocoo.org/community/poweredby/`, 2014. [Online; accessed 26-June-2015].

[50] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[51] Kush R. Varshney, Dennis Wei, Karthikeyan Natesan Ramamurthy, and Aleksandra Mojsilović. Data challenges in disease response: The 2014 ebola outbreak and beyond. *J. Data and Information Quality*, 6(2-3):5:1–5:3, June 2015.

[52] 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.

[53] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[54] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[56] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, October 2012.

[57] Judicael A. Zounmevo, Dries Kimpe, Robert Ross, and Ahmad Afsahi. Using mpi in high-performance computing services. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 43–48, New York, NY, USA, 2013. ACM.