ASSURED INFORMATION SHARING (AIS) USING THE CLOUD

by

Vaibhav Khadilkar

APPROVED BY SUPERVISORY COMMITTEE:

__________________________
Dr. Bhavani Thuraisingham, Co-Chair

__________________________
Dr. Murat Kantarcioglu, Co-Chair

__________________________
Dr. Latifur Khan

__________________________
Dr. Farokh Bastani
This dissertation

is dedicated to my parents.
ASSURED INFORMATION SHARING (AIS) USING THE CLOUD

by

VAIBHAV KHADILKAR, BS, MS

DISSERTATION
Presented to the Faculty of
The University of Texas at Dallas
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS
December 2013
ACKNOWLEDGMENTS

I am deeply grateful to my advisors, Dr. Bhavani Thuraisingham and Dr. Murat Kantarcioglu, for supporting me throughout my graduate school career. They have not only provided me with numerous research opportunities and economic support, but have also allowed me great freedom to pursue independent work. Under their guidance, I have learnt how to think about research problems, and that designing and developing practically usable systems is as necessary as conducting theoretical research.

I am also especially grateful to Dr. Latifur Khan and Dr. Farokh Bastani for agreeing to serve on my dissertation committee. Their comments on my work have been invaluable and have allowed me to greatly improve this dissertation.

I am greatly indebted to my parents, not only for their patience and understanding throughout my educational endeavors, but also for giving me the freedom to make my own decisions since childhood. I would not be where I am without their sacrifice and support.

I also gratefully acknowledge the institutional support I have received to undertake this dissertation, without which this work would not be possible. This work was partially supported by The Air Force Office of Scientific Research (AFOSR) MURI Grant FA9550-08-1-0265 and FA9550-12-1-0082, The National Institutes of Health (NIH) Grants 1R01LM009989 and 1R01HG006844, The National Science Foundation (NSF) Grants Career-CNS-0845803, CNS-0964350, CNS-1016343, CNS-1111529, CNS-1228198 and the Army Research Office (ARO) Grant W911NF-12-1-0558. I thank Dr. Robert Herklotz for his support.

October 2013
PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the “Guide for the Preparation of Master’s Theses and Doctoral Dissertations at The University of Texas at Dallas.” It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student’s contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.
Assured Information Sharing (AIS) refers to the ability of multiple organizations to securely share information. Given the precarious times we live in today, a lack of information sharing, particularly between government agencies, could lead to incidents that cause irreparable damage to life and property. A neglected area in the context of AIS is technological infrastructure, which refers to the software ecosystem that allows organizations to efficiently, economically and securely share information. Although researchers have developed policy-based information sharing systems, none of these are cloud-based; therefore, they lack the scalability/efficiency needed for supporting a large user-base that utilizes vast quantities of data. In this dissertation, we try to remedy this situation by developing AIS implementations that operate under varying conditions.

The first part of this dissertation presents details of two prototypes that provide data sharing and analysis capabilities for a significant number of users. The first prototype, CAISS-X, uses a cloud framework for managing relational data and a non-cloud policy engine to enforce XACML policies. Although CAISS-X represents a discernible enhancement over prior AIS
implementations, it still suffers from constraints related with usage of relational model and XACML-based policies. The second prototype, CAISS, overcomes these limitations by using a cloud framework for data storage/retrieval and policy enforcement.

The next part of this dissertation presents Hybridizer, a framework that allows organizations to automatically partition their data/processing tasks over hybrid clouds, while taking into account performance, security and financial requirements. Hybridizer allows organizations to fully exploit the benefits of hybrid clouds towards developing AIS solutions, while achieving the right mix of performance, security and financial costs.

The final part of this dissertation describes StormRider, a system that allows organizations to securely manage large-scale, evolving networks in real-time. StormRider stores/queries network data using RDF/SPARQL due to their expressivity and ability to capture evolving domain requirements. Since there did not exist a comprehensive, distributed RDF storage framework, the last part also describes Jena-HBase, a framework that uses existing cloud technologies to construct a distributed, scalable and efficient RDF storage framework.
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** ................................................................. v  
**PREFACE** ........................................................................... vi  
**ABSTRACT** ......................................................................... vii  
**LIST OF FIGURES** ............................................................. xiii  
**LIST OF TABLES** ............................................................... xvi  
**CHAPTER 1 INTRODUCTION** .................................................. 1  
  1.1 Assured Information Sharing (AIS) ......................................... 1  
     1.1.1 Requirements and Architecture .................................... 2  
     1.1.2 Motivation .................................................................. 5  
     1.1.3 Challenges .................................................................. 8  
  1.2 Assured Information Sharing (AIS) using the Cloud ................. 10  
     1.2.1 AIS using Private Clouds ............................................ 10  
     1.2.2 AIS using Hybrid Clouds ........................................... 11  
     1.2.3 Real-Time AIS .......................................................... 11  
     1.2.4 Our Contributions ..................................................... 12  
     1.2.5 Dissertation Roadmap ............................................... 13  
**CHAPTER 2 BACKGROUND INFORMATION** ......................... 15  
  2.1 Overview of Semantic Web Technologies ............................... 15  
     2.1.1 Overview of RDF ..................................................... 15  
     2.1.2 Overview of SPARQL ............................................... 16  
     2.1.3 Overview of RDFS and OWL ................................. 17  
     2.1.4 Overview of Jena ................................................... 18  
  2.2 Overview of Cloud Computing Technologies .......................... 20  
     2.2.1 Overview of Apache Hadoop .................................... 20  
     2.2.2 Overview of Apache HBase ..................................... 21
LIST OF FIGURES

1.1 AIS - Goal-oriented Architectural Overview ........................................... 3
2.1 Jena Architectural Overview ................................................................. 19
3.1 CAISS-X Architecture ........................................................................... 31
3.2 CAISS Architecture .............................................................................. 37
4.1 Architectural Overview of Hybridizer ..................................................... 63
4.2 Run-Time Component of Hybridizer ....................................................... 65
4.3 Performance of Dynamic Programming (DP) vs. CPLEX Optimizer for the Outsourcing $WDP$ variant for PD .......................................................... 71
4.4 Performance of Dynamic Programming (DP) vs. CPLEX Optimizer for the Cloudbursting $WDP$ variant for PD2 .................................................. 73
4.5 Performance of Dynamic Programming (DP) vs. CPLEX Optimizer for the Hybrid $WDP$ variant for PD2 ........................................................... 74
5.1 StormRider – Architectural Overview ....................................................... 79
5.2 HBase View: Node-centric information .................................................. 82
5.3 HBase View: Landmark-centric information ........................................... 83
5.4 A sample Add-Topology for the Twitter network ..................................... 88
5.5 A Query-Topology used to execute SPARQL queries ............................... 98
5.6 A sample user-defined Query-Topology ................................................ 106
5.7 An Analyze-Topology used to compute centrality metrics on the Twitter network ................................................................. 108
5.8 Performance of StormRider for Closeness Centrality computation ........... 116
5.9 Performance of StormRider for Betweenness Centrality computation .......... 117
5.10 Overhead of enforcing redaction policies in StormRider ......................... 119
6.1 Jena SDB – Architectural Overview ....................................................... 124
6.2 Jena-HBase – Architectural Overview.................................................... 126
6.3 Simple Layout – Storage Schema ............................................................ 127
6.4 Vertically Partitioned Layout – Storage Schema ...................................... 128
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Indexed Layout – Storage Schema</td>
<td>129</td>
</tr>
<tr>
<td>6.6</td>
<td>Vertically Partitioned and Indexed Layout – Storage Schema</td>
<td>130</td>
</tr>
<tr>
<td>6.7</td>
<td>Hybrid Layout – Storage Schema</td>
<td>132</td>
</tr>
<tr>
<td>6.8</td>
<td>Hash Layout – Storage Schema</td>
<td>133</td>
</tr>
<tr>
<td>6.9</td>
<td>Comparison of Jena-HBase layouts for SP²Bench – Graph Loading</td>
<td>143</td>
</tr>
<tr>
<td>6.10</td>
<td>Comparison of Jena-HBase layouts for SP²Bench – Graph Querying (S denotes an HBase scanner timeout)</td>
<td>144</td>
</tr>
<tr>
<td>6.11</td>
<td>Comparison of Jena-HBase layouts for LUBM – Graph Loading</td>
<td>146</td>
</tr>
<tr>
<td>6.12</td>
<td>Comparison of Jena-HBase layouts for LUBM – Graph Querying</td>
<td>148</td>
</tr>
<tr>
<td>6.13</td>
<td>Comparison of Jena TDB vs. Jena-HBase Hybrid layout for SP²Bench – Graph Loading (M denotes an Out Of Memory exception)</td>
<td>151</td>
</tr>
<tr>
<td>6.14</td>
<td>Comparison of Jena TDB vs. Jena-HBase Hybrid layout for SP²Bench – Graph Querying (M denotes an Out Of Memory exception while T denotes an Execution timeout)</td>
<td>152</td>
</tr>
<tr>
<td>6.15</td>
<td>Comparison of Jena TDB vs. Jena-HBase Hybrid layout for LUBM – Graph Loading (M denotes an Out Of Memory exception)</td>
<td>154</td>
</tr>
<tr>
<td>6.16</td>
<td>Comparison of Jena TDB vs. Jena-HBase Hybrid layout for LUBM – Graph Querying (M denotes an Out Of Memory exception)</td>
<td>155</td>
</tr>
<tr>
<td>B.1</td>
<td>Jena-MM Storage Architecture</td>
<td>198</td>
</tr>
<tr>
<td>B.2</td>
<td>Lucene model layout</td>
<td>200</td>
</tr>
<tr>
<td>B.3</td>
<td>Query Processing in Jena-MM</td>
<td>203</td>
</tr>
<tr>
<td>B.4</td>
<td>Comparison of LRU-based algorithms for SP²Bench: Queries Q1–Q6</td>
<td>208</td>
</tr>
<tr>
<td>B.5</td>
<td>Comparison of LRU-based algorithms for SP²Bench: Queries Q7–Q12</td>
<td>209</td>
</tr>
<tr>
<td>B.6</td>
<td>Comparison of MRU-based algorithms for SP²Bench: Queries Q1–Q6</td>
<td>210</td>
</tr>
<tr>
<td>B.7</td>
<td>Comparison of MRU-based algorithms for SP²Bench: Queries Q7–Q12</td>
<td>211</td>
</tr>
<tr>
<td>B.8</td>
<td>Comparison of all algorithms for SP²Bench: Queries Q1–Q6</td>
<td>213</td>
</tr>
<tr>
<td>B.9</td>
<td>Comparison of all algorithms for SP²Bench: Queries Q7–Q12</td>
<td>214</td>
</tr>
<tr>
<td>B.10</td>
<td>Comparison of LRU-based algorithms for LUBM: Queries Q1–Q14</td>
<td>216</td>
</tr>
<tr>
<td>B.11</td>
<td>Comparison of MRU-based algorithms for LUBM: Queries Q1–Q14</td>
<td>217</td>
</tr>
<tr>
<td>B.12</td>
<td>Comparison of all algorithms for LUBM: Queries Q1–Q14</td>
<td>218</td>
</tr>
<tr>
<td>B.13</td>
<td>Comparison of persistence strategies for SP²Bench: Queries Q1–Q6 (M denotes an Out Of Memory exception)</td>
<td>220</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>B.14 Comparison of persistence strategies for SP²Bench: Queries Q7–Q12</td>
<td>221</td>
<td></td>
</tr>
<tr>
<td>(M denotes an Out Of Memory exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.15 Comparison of persistence strategies for LUBM: Queries Q1–Q14</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>(T denotes a Timeout exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.16 Comparison of SP²Bench Q3a and Q8 for various approaches</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>(M denotes an Out Of Memory exception)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B.17 Comparison of LUBM Q6 and Q13 for various approaches</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>(M denotes an Out Of Memory exception)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Design Criteria and their Applicability to Cloud Models</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Notations used within the Task Partitioning Problem</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Specification of Parameters for PD</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Queries and Redaction Operations used in Experimental Evaluation; ST denotes Sanitized Triples</td>
<td>120</td>
</tr>
<tr>
<td>A.1</td>
<td>Summary of Statistics for RDF Data</td>
<td>180</td>
</tr>
<tr>
<td>A.2</td>
<td>Summary of Selectivity Estimation for RDF</td>
<td>184</td>
</tr>
<tr>
<td>B.1</td>
<td>Variables used in Jena-MM</td>
<td>202</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

One of the keys to effectively combat terror lies in the ability to share critical information that has been collected by multiple cooperating agencies (national and international). The process of data sharing, in this context, has several real-world implications, the foremost of which, is the ability to make proactive decisions that could help save thousands of lives. However, the process must ensure that sensitive information is protected and the privacy of individuals is preserved. Additionally, the process must also be able to integrate unanticipated, variable data sets that arise from the ever-changing conditions in which agencies operate.

The Assured Information Sharing (AIS) framework is envisioned as a solution that addresses various challenges integral to the process of data sharing. Our focus is on one of the main challenges that hinders secure data sharing, namely access to secure technological infrastructures that operate under varying conditions, which facilitate secure data sharing and analysis by a significant number of users. We begin by presenting an overview of AIS; this includes requirements and a goal-oriented architecture of AIS, the necessity of AIS implementations, and some of the challenges associated with adoption of AIS. This is followed by a brief discussion of various secure technological infrastructures provided in this dissertation.

1.1 Assured Information Sharing (AIS)

In this section, we first present the requirements and architecture of an AIS framework that meets the DoD AIS requirements.\footnote{We acknowledge that a part of the contents of this chapter will be published in Khadilkar et al., 2014, which is available at: \url{http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-1-4614-3295-1} The final publication is available at: \url{link.springer.com}} Secondly, we present some motivating factors that drive
the design of an AIS framework. Finally, we present some challenges that must be overcome before reaching our goal of having an effective process for assured information sharing.

1.1.1 Requirements and Architecture

There are several components that make up an AIS framework. However, the core of an AIS framework is made up of the following two themes that cut across all components:

- **Agile environment**: The ability to rapidly adapt to changing ground conditions, thus ensuring that an implementation can readily integrate data created due to unexpected partnerships or events, which can then be quickly converted into actionable intelligence.

- **Security**: The capacity to protect sensitive information, preserve the privacy of individuals, verify the integrity of data, enforce access control mechanisms when retrieving data as well as controlling how data is used upon release by a user.

We now present two DoD requirements that capture the intuitive concepts involved in the process of assured information sharing:

- Provide the ability to dynamically/securely share information at multiple classification levels among U.S., allied and coalition forces \cite{Anderson2008, DoD-Air-Force2011}.

- Deliver the power of information to ensure mission success through an agile enterprise with freedom of maneuverability across the information environment \cite{US-DoD2007a, DoD-Air-Force2011}.

As should be evident from the above requirements, the key to an AIS framework is to allow high quality information to be shared securely, leading to timely decision making. Towards this end, Figure 1.1 presents a goal-oriented architectural overview of an AIS framework based on the approaches taken to achieve the information sharing goals outlined in the DOD Information Sharing Strategy \cite{US-DoD2007a}. 
The main component of Figure 1.1 is the ongoing cycle that defines the set of activities required to make a collaborative mission successful. The cycle itself is made up of several phases, beginning with *Mission Need*, in which a set of requirements for a particular mission are identified collaboratively by organizations participating in the process of information sharing. This is followed by the *Discover/Collect Information* phase in which co-operating organizations use their ongoing data collection processes to capture data essential to the mission at hand. Then, the *Process* phase performs the task of information assimilation by checking the trustworthiness of data, resolving possible ambiguities in data and structuring...
data for the next phase. In the *Analyze* phase, structured data is mined for possible trends or patterns associated with the current mission. After that, the *Integrate* phase links knowledge extracted from various information sources that is relevant to the objectives of the ongoing mission. The *Inform* phase ensures that knowledge derived from the integrated sources of information is shared with relevant personnel and with organizations that are presently a part of the information sharing process. Finally, the *Act* phase allows participating organizations to operate on actionable intelligence produced by the previous phases. A point to note is that the phases from *Process* to *Inform* are provided with a feedback mechanism through which they can advise their respective previous phases of improvements that can be immediately made so that the overall process is better aligned with the goals of the current mission.

The cyclic structure of the information sharing process allows the results of an iteration to be used for refinement of mission objectives, which are then cascaded over to the remaining phases of the next iteration. Additionally, this ensures that information sharing becomes an ongoing activity in which various organizations can join/leave the process based on the requirements of a specific mission that they want to carry out. Finally, the phases defined in the cyclic information sharing process represent a “*Discovery to Decision*” continuum that captures the notion of continued collaboration among participating organizations towards the creation of timely and actionable intelligence (US-DoD, 2007a).

The phases between *Discover/Collect Information* and *Inform* collectively create the concept of “*Information Mobility*”. This idea captures the readiness required within an organization to address predefined mission requirements as well as unforeseen requirements of unanticipated partners and events. Consequently, information mobility promotes a higher degree of intra- and inter-organizational situational awareness among cooperating partners. Finally, information mobility can be achieved through the use of appropriate technological tools (*viz.* enterprise architectures, services and communications infrastructure), trust building/maintenance tools (*viz.* training exercises and social networking) and security tools (*viz.* intra- and inter-domain policies that address integrity, confidentiality, *etc.*).
The outer layer of Figure 1.1 shows $n$ different organizations participating in the information sharing process. Each of these organizations is an autonomous entity and as such defines its own Management structure. This structure allows an organization to effectively streamline current operational budgets towards the task of information sharing. Additionally, an organization should be provided with incentives in future budgets (for e.g., through federal funding) to encourage information sharing since it leads to higher quality data, which in turn leads to informed decisions. The collaboration between participating organizations also leads to the creation of a Federated Information Sharing environment. The federated approach strengthens relationships between collaborating organizations and creates an information sharing framework that easily incorporates trust mechanisms, best-of-breed technological tools and compliance procedures. In addition, federation allows organizations to support an agile environment that is conducive towards rapid integration of unanticipated partners and events. Finally, the federated approach also creates the notion of Information as a force multiplier, since when a large amount of information is shared, it can become a force if it is used appropriately while verifying its accuracy, consistency, authority and completeness.

As stated earlier, the two themes of agile environment and security cut across all phases and goals described in Figure 1.1. The agility aspect refers to the ability of an organization to dynamically handle the data and requirements of unanticipated partners and events and produce timely decisions. The security aspect refers to the capability of an AIS framework to ensure that information is protected at all levels (viz. intra- and inter-organization) through the use of appropriate policies and procedures.

1.1.2 Motivation

In this subsection, we present some of the motivating factors that necessitate the development of an AIS framework. First, we present two motivating scenarios in which co-operating agencies would greatly benefit from the use of an AIS framework. This is followed by a
general discussion of the basic reasons that compel the immediate development of various AIS implementations.

Motivating Scenarios: As stated earlier, an AIS framework provides a mechanism through which multiple, co-operating agencies are able to dynamically and securely share information.

Scenario 1: Given the precarious times we live in today, a lack of information sharing between organizations, particularly government agencies, could lead to incidents that cause irreparable damage to life and property (Newton-Small 2013; Rosen and Sellmyer 2013; FoxNews.com 2013). In the case of the Boston Marathon bombings, hearings examining the bombings clearly suggest that there was a lack of information sharing about the instigators between federal, state and local authorities prior to the attack (Rosen and Sellmyer 2013). In a separate incident, agencies in the Justice Department did not share the new identities of known or suspected terrorists in witness protection, which in turn meant that the new identities were not updated on no-fly lists, thereby allowing them to board commercial flights (FoxNews.com 2013). To eliminate such intelligence holes, it is necessary to develop various AIS implementations that operate under varying conditions.

Scenario 2: A relevant use case where an AIS framework would provide significant value-added benefits pertains to the Distributed Common Ground System (DCGS) (Lockheed-Martin 2012). The DCGS is a global, internet-like network where both military and national agencies have access to time sensitive intelligence, surveillance and reconnaissance (ISR) data. The main motivation behind the DCGS project is to provide real-time access to actionable intelligence to the various U.S. combat forces that are deployed throughout the world. Each of the three military departments, namely the Army, the Navy and the Air Force have developed their own versions of the DCGS (DoD-Air-Force 2009) which are interfaced through the DCGS Integration Backbone (DIB) (DoD-Air-Force 2012). The DIB provides the architecture, standards, tools, and documentation for connecting current and future systems to the Global Information Grid (GIG) (US-DoD 2007b).
An AIS implementation could be built on top of the DIB software stack that encourages co-operating agencies (military and national) to proactively share information through the use of flexible security mechanisms. Additionally, the implementation would support rapid integration of evolving data sources that arise due to the volatile environment in which agencies function. Finally, the implementation would foster trusted relationships between various stakeholders while allowing each of them to operate in an agile enterprise environment that accommodates unforeseen circumstances.

Motivating Factors: As illustrated by the scenarios presented, there are several motivating factors that encourage the immediate development of various AIS implementations:

- *Encourage/Incentivize information sharing*: The various coalition partners are encouraged/incentivized to participate in the process of information sharing, which creates a win-win situation for them, since it provides them with higher quality data that can then be used to build finer data analytics tools (*viz.* data mining models, trend analysis, pattern recognition, *etc.*).

- *“Need to Share” over “Need to Know”*: The ability to proactively share information between co-operating agencies through the use of security mechanisms that support the “need to share” philosophy rather than the traditional “need to know” paradigm.

- *Rapid integration of evolving data sources*: The capability of quickly identifying and integrating data from sources that are continuously evolving entities. Note that, data sources could be traditional, such as flat files, spreadsheets, databases, *etc.* or contemporary, such as smartphones, high-resolution cameras, networking equipment, *etc.*

- *Promoting trusted relationships between stakeholders*: The promotion of trusted relationships between the different stakeholders ensured through the use of appropriate trust mechanisms (*viz.* cryptographic mechanisms, access control, redaction, *etc.*).
• **Agile enterprise environment**: The creation of an agile enterprise environment for each stakeholder thus enabling them to adapt to unexpected partners (*e.g.* an international intelligence agency) and events such as an imminent terrorist threat or activity.

1.1.3 Challenges

In this subsection, we outline different challenges associated with the adoption of an AIS framework within an organization. The challenges are based on the touchstones presented in the DoD Information Sharing Strategy ([US-DoD] 2007a) and therefore, are broadly classified into the following five categories:

• **Cultural**: Since the process of information sharing always involves a tradeoff between resource allocation/consumption and security risks, many agencies would be averse to participating in this process. Therefore, it is necessary to change the mindset of an organization from one of information “ownership” to that of information “stewardship”. Consequently, a solution that addresses this category would involve the following open-ended list of activities:

  – Inculcate the information sharing methodology into training curricula.
  
  – Promote an environment for information sharing throughout the organization through the development/maintenance of a risk-based management approach.

• **Policy**: The existing laws, policies, regulations, and rules within an organization may not readily support the process of information sharing. Therefore, it may be necessary to restructure an organization’s internal policies towards the common vision of assured information sharing. As a result, a solution that addresses this category would involve the following open-ended list of activities:

  – Evaluate existing policies within an organization and promote the development/ modification of rules that encourage information sharing.
– Develop techniques that allow sharing information while preserving the confidentiality of sensitive information and respecting the privacy of individuals.

• Governance: The management framework currently in place within an organization may not be equipped with creating/sustaining an information sharing environment. Hence, it may be necessary to modify this framework so that various processes/policies associated with information sharing can be monitored effectively. Additionally, a solution that addresses this touchstone must involve the following, not necessarily complete, list of activities:

  – Develop a management structure within an organization to govern the various activities associated with the process of information sharing.
  – Create/maintain practices within an organization that are aligned with the technical, legal and performance aspects of information sharing.

• Economics and Resources: The economics of moving towards an assured information sharing framework may not be feasible for an organization. Consequently, an organization needs to put in place budgetary planning, management and performance methodologies to ensure that the information sharing process is cost-effective while providing a significant return-on-investment. Some of the activities that are part of a solution that addresses this challenge can be summarized as follows:

  – Inculcate the vision of information sharing into financial processes.
  – Develop/maintain practices as part of the budgeting process to ensure a cost-effective information sharing process.

• Secure Technology Infrastructure: The current technological infrastructure within an organization may not be completely equipped to handle the various components of an AIS framework. Therefore, an organization may require the integration of newly
developed technologies with their existing IT environment to support the information sharing framework. A solution that addresses this challenge would be composed of the following list of activities:

- Ensure that newly developed information sharing technologies are compliant with existing federal IT design and technology standards.
- Enforce the implementation of federally approved security mechanisms within a technology/product/service.

1.2 Assured Information Sharing (AIS) using the Cloud

As stated earlier, the focus of this dissertation is developing secure technological infrastructures that operate under varying conditions, which facilitate secure data sharing and analysis by a significant number of users. In this section, we summarize the secure technological infrastructures, we highlight the contributions of this dissertation and we present a roadmap for the rest of this dissertation.

1.2.1 AIS using Private Clouds

As should be evident from the DoD AIS requirements described earlier, there is an urgent need for the development of efficient and scalable AIS tools and technologies. One recent technology that is capable of handling these requirements is cloud computing, whose adoption by various big businesses has validated its possible advantages of efficiency, flexibility, scalability and cost-effectiveness. Multiple federal agencies have also shifted their focus towards adopting a cloud-centric approach to fulfill their information sharing capabilities (Hoover 2011, US-DoD 2012, 2007a, Defense-Market 2009). Furthermore, even though a number of policy-based information sharing tools have been developed (Rao et al. 2008, Thuraisingham et al. 2008, Finin et al. 2009), none of these operate in a cloud environment, thus being
unable to address the core requirement of large scale data analysis by a significant number of users. The first part of this dissertation presents the details of two prototype systems that address this deficiency. The first system, called CAISS-X, uses a cloud-based framework for managing relational data and a non-cloud policy engine to enforce XACML policies. Although CAISS-X represents a discernible enhancement over prior AIS implementations, it still suffers from constraints due to usage of the relational model and XACML-based policies. The second system, called CAISS, overcomes these limitations by using a cloud-based framework for both, data storage and retrieval as well as policy enforcement.

1.2.2 AIS using Hybrid Clouds

The emergence of hybrid clouds has allowed organizations to develop AIS solutions that completely utilize their in-house IT resources for addressing business critical requirements, while turning to low-cost public cloud services for securely sharing data. However, before an organization can develop AIS solutions over hybrid clouds, they need to have a mechanism that automatically partitions their data and processing tasks over a hybrid cloud, while taking into account organizational performance, security and financial requirements. The next part of this dissertation presents a principled framework, called Hybridizer, to address this challenge for various hybrid cloud deployment models, while ensuring that an organization’s requirements are fulfilled. Hybridizer allows one to fully exploit the benefits of a hybrid cloud, while achieving the right mix of performance, security and financial costs.

1.2.3 Real-Time AIS

In today’s precarious times, it is essential for cooperating federal, state and local agencies to be able to securely share information in real-time. There have been several calls by both, government agencies as well as private companies to address this issue (Albanesius, 2013; Higgins, 2012; Lawton, 2012). Furthermore, the government already uses systems such as
the Distributed Common Ground System (DCGS), which provides real-time access to intelligence, surveillance and reconnaissance data for U.S. combat forces deployed throughout the world [Lockheed-Martin, 2012]. If AIS implementations could be developed over existing frameworks such as DCGS, this would provide secure, real-time information sharing capabilities that produce high quality data, which in turn results in timely decision making. Now, information networks are a major category of data available in the intelligence community, since they can model a number of domains such as communication networks, terrorist cells and information flow networks. To develop an AIS implementation that operates over information networks, one needs to develop an underpinning that allows cooperating parties to securely manage large-scale, evolving networks in real-time. The final part of this dissertation provides such an underpinning, called StormRider, a system that uses existing Cloud Computing and Semantic Web technologies to securely manage evolving networks in real-time. StormRider stores and queries network data using the RDF data model and SPARQL due to their expressivity and ability to capture evolving domain requirements. Since there did not exist a comprehensive, distributed RDF storage framework, the last part of this dissertation also describes Jena-HBase, a framework that uses existing Cloud Computing technologies to construct a distributed, scalable and efficient RDF storage framework.

1.2.4 Our Contributions

In this dissertation, we provide several secure technological infrastructures that operate under varying conditions, thereby addressing one of the core challenges that hinders the adoption of an AIS framework in an organization. In particular, this dissertation provides the following:

- Two prototype systems, namely CAISS-X and CAISS, which use private clouds for managing relational/RDF data, while providing secure access using non-cloud and cloud-based policy engines.
• A principled framework called Hybridizer that formalizes and solves the problem of automatically distributing an organization’s task workload over various hybrid cloud deployment models, while adhering to organizational requirements.

• An underpinning for AIS implementations called StormRider that combines existing Cloud Computing and Semantic Web technologies to provide developers with tools that securely manage large-scale, evolving networks in real-time.

• A storage framework called Jena-HBase that uses existing Cloud Computing technologies to construct a distributed, scalable and efficient RDF storage framework.

1.2.5 Dissertation Roadmap

The rest of this dissertation is organized as follows:

• Chapter 7 surveys the significant body of existing research work that has contributed towards the development of secure technological infrastructures for various IT environments that are presented in this dissertation.

• Chapter 2 provides an overview of Semantic Web and Cloud Computing technologies employed in the development of various secure technological infrastructures outlined in this dissertation. Furthermore, the chapter presents a summary of access control models/languages used in this dissertation.

• Chapter 3 presents details about two prototype systems, namely CAISS-X and CAISS, which use private clouds to provide AIS capabilities. CAISS-X uses a cloud-based framework for managing data and a non-cloud policy engine to enforce XACML policies, while CAISS uses a cloud-based framework for both, data storage and retrieval as well as policy enforcement.
• Chapter 4 provides details about *Hybridizer*, a hybrid cloud framework that allows organizations participating in the AIS process to automatically partition their data and processing tasks over a hybrid cloud, while taking into account organizational performance, security and financial requirements.

• Chapter 5 presents StormRider, an underpinning for AIS implementations that allows secure, real-time information sharing by combining existing Cloud Computing and Semantic Web technologies to provide developers with tools that securely store, query and analyze large-scale, evolving networks in real-time.

• Chapter 6 presents details about Jena-HBase, a framework that uses existing Cloud Computing technologies to construct a distributed, scalable and efficient RDF storage framework. Jena-HBase is used by StormRider to store/query network data in real-time using RDF/SPARQL.

• In Chapter 8 we present a broad set of conclusions that can be drawn from this dissertation and we also provide some directions for future research.
CHAPTER 2
BACKGROUND INFORMATION

In this chapter, we provide an overview of Semantic Web and Cloud Computing technologies employed in the development of various systems outlined in this dissertation. Furthermore, we present a summary of access control models/languages used in this dissertation.

2.1 Overview of Semantic Web Technologies

2.1.1 Overview of RDF

The Resource Description Framework (RDF) is a model used to represent metadata about data on the Web. RDF can not only be used to represent metadata about Web resources but also to represent information about resources themselves. RDF defines a graph data model to capture relationships between resources using the concept of a triple. A triple comprises a subject, predicate and object; a set of triples forms an RDF graph. A triple asserts that the relationship represented by the predicate is held between the subject and object.

RDF is extremely flexible since a resource is not constrained by its type or associated properties. Moreover, RDF uses the open-world assumption that allows anyone to make statements about any resource. Furthermore, RDF differs from other data models in that it does not depend on fixed schemas or type hierarchies.

The concepts underlying RDF can be specified more formally as follows:

Definition 2.1. The RDF data model is composed of a set of terms, $T$, which comprises pairwise disjoint infinite sets, $I$, $B$ and $L$ (IRI’s, Blank nodes and Literals respectively).

---

1. http://www.w3.org/RDF/
Definition 2.2. A triple \((s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)\), where \(s\) denotes a subject, \(p\) denotes a predicate, and \(o\) denotes an object.

2.1.2 Overview of SPARQL

SPARQL\(^2\) is a query language for RDF that uses pattern matching to match a subgraph of an RDF graph. SPARQL can be used to design queries over disparate data sources, where a source contains native RDF data or data viewed as RDF via middleware. In the following, we list the basic formalisms of SPARQL.

Definition 2.3. A triple pattern, \(TP\), is a triple of the form \((sp, pp, op) \in (I \cup V \cup L \cup B) \times (I \cup V) \times (I \cup V \cup L \cup B)\), where \(V\) is an infinite set of variables that is pairwise disjoint from sets \(I\), \(B\), and \(L\), while \(sp, pp\), and \(op\) denote a subject pattern, predicate pattern and object pattern respectively.

Definition 2.4. A mapping \(\mu\) is a partial function \(\mu : V \rightarrow T\). Given a triple pattern \(t\), \(\mu(t)\) is the triple obtained by replacing variables in \(t\) according to \(\mu\) (Pérez et al., 2006). Domain of \(\mu\), \(\text{dom}(\mu)\) is the subset of \(V\) where \(\mu\) is defined and \(\Omega\) is defined as a set of mappings \(\mu\) (Pérez et al., 2006).

Definition 2.5. A basic graph pattern, \(BGP\), is a finite set of triple patterns.

Definition 2.6. A graph pattern, \(GP\), is defined recursively as follows (Pérez et al., 2006):

1. A basic graph pattern is a graph pattern.
2. If \(P_1\) and \(P_2\) are graph patterns, then \((P_1 \ \text{AND} \ P_2)\), \((P_1 \ \text{UNION} \ P_2)\) and \((P_1 \ \text{OPT} \ P_2)\) are graph patterns.
3. If \(P\) is a graph pattern and \(X \in (I \cup V)\), then \((X \ \text{GRAPH} \ P)\) is a graph pattern.
4. If \(P\) is a \(GP\) and \(R\) is a value constraint, then \((P \ \text{FILTER} \ R)\) is a graph pattern.

\(^2\)http://www.w3.org/TR/sparql11-query/
2.1.3 Overview of RDFS and OWL

RDF provides a simple way to represent information about resources (viz. subjects) using named properties (viz. predicates) and values (viz. objects). However, users may need to define application-specific classes and properties to construct triples. Since RDF does not allow the definition of custom vocabularies, the World Wide Web Consortium (W3C) has developed two languages, RDF Schema (RDFS) and Web Ontology Language (OWL), which can be used to develop custom vocabularies. We now give a brief overview of these languages.

Overview of RDF Schema (RDFS)

RDF Schema (RDFS) provides a user with tools to describe custom classes and properties. Additionally, a user can also specify which classes and properties should be used together. For example, a user can say that property `ex:studentId` will be used in describing a `ex:Student`. In this sense, RDFS is similar to type systems found in object-oriented programming languages. However, RDFS does not mandate that information must be placed into defined classes and properties like programming language types, but instead allows a user to use these constructs to provide additional information about resources. Finally, the facilities provided by RDFS are predefined in a specialized set of RDF resources with their own semantics.

A class in RDFS is used to identify types of resources that need to be described, and can be defined using resources `rdfs:Class` and `rdfs:Resource` and properties `rdf:type` and `rdfs:subClassOf`. These terms allow a user to define both, class membership for resources and a type hierarchy for classes. In addition to defining classes, a user must also be able to define properties that allow them to specify additional features for classes. In RDFS, properties can be defined using the RDF class `rdf:property` and the properties `rdfs:domain`, `rdfs:range` and `rdfs:subPropertyOf`. Again, it should be evident that in addition to defining properties,
a user can construct a hierarchy of properties (using rdfs:subPropertyOf) as well as provide instructions on how classes and properties are to be used together in RDF data.

**Overview of Web Ontology Language (OWL)**

The Web Ontology Language (OWL) extends the vocabulary defined in RDFS for describing custom-built vocabularies to include relationships between classes, cardinality restrictions, more expressive types of properties, property characteristics, and enumerated classes. The OWL vocabulary has a formal semantics based on description logics, which are a decidable fragment of first order logic. Also, OWL provides users with three increasingly expressive sublanguages to use based on their specific requirements, as summarized in footnote 4.

1. **OWL Lite** supports use cases requiring a classification hierarchy and simple constraints such as permitting cardinality constraints having only values 0 or 1.

2. **OWL DL** supports use cases requiring the maximum expressiveness, while at the same time being computationally complete and decidable. OWL DL includes all constructs in OWL, however, one is able to use them only under certain restrictions.

3. **OWL Full** supports use cases requiring the highest form of expressiveness, without retaining any computational guarantees.

### 2.1.4 Overview of Jena

Figure 2.1 provides an overview of Jena which provides a rich set of features for interacting with RDF graphs. At its core, Jena uses the RDF API to allow one to create and query graphs. At this level, one interacts with Models that allow an English-like representation

[http://www.w3.org/TR/owl-features/](http://www.w3.org/TR/owl-features/)

of triples. In addition, the RDF API allows users to read RDF data from external sources (files or URL’s) and to de-serialize/serialize graphs in the most common RDF syntaxes.

While the Model interface provides a high-level abstraction for users, internally an RDF graph is stored in a more simpler abstraction called Graph. This allows Jena to use various storage mechanisms such as in-memory, relational database (SDB) or a custom-built disk-based tuple index (TDB) equivalently. Additionally, the Graph interface also provides a plug-in point for users to connect their own customized triple stores with Jena.

The Inference API provides the ability to infer additional information that is not stated in a graph by using different rulesets (built-in or custom-defined). In addition, the inference API also allows external reasoners to be connected with Jena for performing inference.

The SPARQL API allows users to query RDF graphs through the SPARQL query language. Jena supports both, the query and update functionalities of SPARQL and also tracks the revisions and updates in the areas of SPARQL that are currently in-development.
The Ontology API provides different methods for interacting with the two richer ontology languages, namely RDFS and OWL. These languages capture the real-world domain that an application tries to model in a logical description.

Finally, Fuseki acts as a data publishing server, thus allowing RDF graphs to be accessed and updated using SPARQL and HTTP.

2.2 Overview of Cloud Computing Technologies

2.2.1 Overview of Apache Hadoop

Apache Hadoop is a framework that allows batch processing tasks to be performed on vast quantities of data. Hadoop uses the Hadoop Distributed File System (HDFS), a distributed file system based on the Google File System (Ghemawat et al., 2003), as its underlying storage mechanism. HDFS provides several advantages such as data replication and fault tolerance. HDFS uses a master/slave architecture comprising a single namenode process (running on the master node) and several datanode processes (usually one per slave node). The namenode manages the file system by tracking how blocks of individual files are distributed over all datanodes. A datanode process is used to keep track of file blocks stored on that node.

Hadoop uses the MapReduce paradigm (Dean and Ghemawat, 2004) to perform batch processing tasks over data stored in HDFS in a parallelized fashion. MapReduce consists of a single JobTracker process (running on the master node) and several TaskTracker processes (usually one per slave node). The JobTracker is responsible for scheduling a MapReduce job among the slave nodes while a TaskTracker is responsible for controlling the execution of the sub-task given to its hosting slave. A MapReduce job consists of three phases: (1) A Map phase in which each slave node performs some computation on data blocks of the input that it has stored. The output of this phase is a key-value pair based on the computation
performed. (2) An intermediate **Sort** phase in which the output of Map phase is sorted based on keys. (3) A **Reduce** phase in which a reducer aggregates various values for a shared key and then further processes them before producing the desired result.

### 2.2.2 Overview of Apache HBase

Apache HBase[^7] is a column-oriented store modeled after Google’s BigTable (Chang et al., 2006). HBase allows the creation and storage of very large tables – billions of rows × millions of columns. It provides several advantages such as query predicate push down and optimizations for real-time queries. HBase uses HDFS as the storage system for its tables. As a result, HBase inherits Hadoop’s master/slave configuration comprising a single master node that coordinates the distribution of data among region servers (usually one per slave).

An HBase table is composed of column families and rows; these may be separated over region servers. A row is defined by a row key and all rows are stored based on a lexicographic sorting of row keys. A column family contains a collection of columns that share the same prefix. For example, columns person:name and person:age are both columns of the family person. The colon (:) character delimits a column family from a column. A column family is defined during schema creation whereas columns can be declared on the fly. A cell is identified as a three tuple: \{row, column, version\}, where the version allows one to store multiple versions of a column value for the same row key. Finally, HBase requires a periodic “commit” operation to reflect any changes (viz. inserts, deletes or updates) to a table.

### 2.2.3 Overview of Apache Hive

Apache Hive[^8] is a data warehousing framework that allows users to manage and analyze datasets stored in HDFS or HBase (Thusoo et al. 2009). Hive provides tools to perform

[^7]: http://hbase.apache.org/
[^8]: http://hive.apache.org/
Extract-Transfer-Load (ETL) operations over data, project structure onto extracted data and query structured data using a SQL-like language called HiveQL. Hive performs query execution using MapReduce while allowing programmers to plug-in custom-built MapReduce programs to perform analytics not supported by HiveQL. Some of the design goals of Hive include dynamic scale-out, user defined analytics and loose coupling with input formats.

The essence of Hive is the ability to structure data as well as to conduct sophisticated analyses of data using HiveQL. Hive reuses the concepts of databases, tables, partitions and data types defined by relational databases to impart structure to data stored in HDFS. In particular, Hive uses the Data Definition Language (DDL) component of HiveQL for managing these tasks. Additionally, Hive uses the Data Manipulation Language (DML) component of HiveQL to query data stored in tables or partitions. A few of the query operations include: selecting only particular columns in a table, selecting rows in a table that match given filters, performing an equi-join operation between two tables, etc.

2.2.4 Overview of Storm

Storm\(^9\) is a distributed real-time computation system. Storm is similar to Hadoop in that it provides users with a framework for performing computations in real-time, much like Hadoop provides users with a framework for performing batch processing operations. Storm provides the following key properties: (1) Support for a broad range of use cases such as stream processing and continuous computation. (2) Storm is scalable and is able to process a large number of messages per second. (3) Storm guarantees that every message will be processed and thereby ensures that there is no data loss. (4) Storm clusters are easily manageable and extremely robust. (5) Storm ensures fault-tolerance by automatically reassigning tasks that fail during execution. (6) Storm’s components are programming language agnostic and therefore Storm can be utilized by nearly anyone.

\(^9\)https://github.com/nathanmarz/storm
A Storm cluster comprises a single **master** node and multiple **worker** nodes. The master runs a Nimbus process that distributes code around the cluster, assigns tasks to machines and monitors for failures. Every worker runs a Supervisor process that listens for work assigned to it by Nimbus and starts/stops worker processes to accomplish this work. The coordination between Nimbus and Supervisors is done through a ZooKeeper\(^\text{10}\) cluster.

Storm uses the concept of a topology to perform real-time computation. A Storm topology is analogous to a MapReduce job, however, a key difference is that a MapReduce job eventually finishes while a topology runs forever or until it is killed. A topology is a directed graph of spouts and bolts connected together with stream groupings. A stream is the core abstraction in Storm and represents an unbounded sequence of tuples that is created and processed in a parallel fashion. A spout acts as a source of streams and usually reads tuples from an external source and emits them into the topology. A bolt is used to perform the desired processing in a topology such as initiating connections with a database, performing operations such as filtering, aggregations and joins. A stream grouping defines how a stream that is input to a bolt is partitioned between that bolt’s tasks. Finally, Storm guarantees that every tuple will be fully processed by the topology and also provides special configurations for customizing the behavior of the nimbus, supervisors and running topologies.

### 2.3 Overview of Access Control Models/Languages

The phenomenon of access control is pervasive; it is found in everyday tasks such as gaining access to a room/building using a keycard as well as in computer-based systems for allowing only authorized users to access a particular file within a file system.\(^\text{11}\) Moreover, it is critical

\(^{10}\)http://zookeeper.apache.org/

\(^{11}\)We acknowledge that the contents of this section will be published in [Khadilkar et al., 2014](http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-1-4614-3295-1), which is available at: [link.springer.com](http://link.springer.com)
for organizations to put in place appropriate access control mechanisms without which disclosure of sensitive data could lead to loss of competitive advantage or damaging lawsuits. We first present a historical perspective on access control models for which an excellent survey is presented in \cite{BertinoSandhu2005}. In addition, we briefly mention some languages that have been developed for the construction and enforcement of access control policies.

### 2.3.1 Overview of Access Control Models

Initial work in the area of access control models focused on developing models based on discretionary access control and mandatory access control policies for relational databases.

**Discretionary Access Control Models:** In models based on discretionary access control (DAC) policies, a user’s access to a resource is based on their identity and authorization rules implemented in the system. Furthermore, these models are discretionary since they allow a user to grant authorizations on data items that they own to other users. One of the first implementations of DAC models was made in the context of relational databases for the System R DBMS \cite{Fagin1978,GriffithsWade1976,Astrahanetal1976}. In this model, tables and views form the objects that need to be protected while SQL functions that can be executed on tables form the set of permissible operations. The enforcement of policies is achieved by looking at ownership of a table and the possible chain of privileges that may be delegated to allow other users to access that table. This model was further enhanced with various features that allow it to better handle modern application requirements through the use of more expressive authorization models. These enhancements include:

- **Negative authorizations:** In this extension, a user can explicitly “deny” others the right to perform certain operations on a table owned by that user \cite{Bertinoetal1997}.

- **Temporal authorizations:** This extension allows users to access tables/views for a specified time period after which their authorizations are revoked \cite{Bertinoetal1998}.
• **Context-aware authorizations**: In this model, user queries against tables/views are rewritten to incorporate contextual information (e.g. time, location, organizational position, etc.) that is part of access control policies (Bacon et al., 2002; Oracle, 2002).

• **Content-Based Access Control**: This extension uses views, which can be defined using queries on tables, for the purpose of defining access control policies (Bertino and Haas, 1988). Then, a query on a view is rewritten by appending to the original WHERE clause, the WHERE clause of the view definition query. The effect of this mechanism is that tuples that do not match predicates in the view definition query are filtered out, thus providing an efficient way to limit access to data.

• **RBAC Model**: This model uses the notion of a role as a bridge between users and authorizations. A role incorporates all necessary authorizations needed to perform a specific function within an organization. In addition, users are assigned to various roles thus allowing them to gain authorizations associated with those roles and thereby perform the required operations on protected resources (Ferraiolo and Kuhn, 1992; Sandhu et al., 1996). The RBAC model has been extended along the following dimensions: the development of temporal constraints in RBAC, called the TRBAC model (Bertino et al., 2001; Joshi et al., 2005), the development of administration models (Crampton and Loizou, 2003; Kern et al., 2004; Koch et al., 2004), the construction of security analysis techniques (Li and Tripunitara, 2004), and more recently, the development of RBAC in a semantically rich language (OWL) (Finin et al., 2008).

**Mandatory Access Control Models**: In models based on mandatory access control (MAC) policies, a user’s access to resources is determined using a predefined classification of users and resources in the system. The classification is made on the basis of a set of access classes defined over all users and resources in the system. Then, a user can access a specific resource only if an ordered relationship exists between the access classes for the user and
the resource. A well known example of this mechanism is based on the Bell and LaPadula model (Bell and LaPadula, 1976) wherein a user can only read resources whose access classes are dominated by the user’s access class (No read-up) and a user can only write resources whose access classes dominate the access class of the user (No write-down). There has also been a lot of research work devoted to addressing some of the issues posed by MAC policies in the context of multilevel secure DBMS’s (Bertino et al., 2001; US-DoD, 1975; Moss, 1985).

2.3.2 Overview of Access Control Languages

The general goals for an access control language are to enable the creation and enforcement of access control policies. At a general level, these languages make assertions about users and their permissible operations (Samarati and di Vimercati, 2000). Additionally, languages have been defined for making authorization decisions for policies that have been combined from various sources (Bonatti et al., 2002) as well as for supporting trust management (Bonatti and Samarati, 2002; Seamons et al., 2002; Yu et al., 2003). Furthermore, since the advent of XML and its acceptance as a de facto standard for information exchange, a number of access control languages based on XML have been proposed (Damiani et al., 2002; Kudoh et al., 2002). A universally accepted XML-based access control language is XACML\footnote{http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-en.html} that allows the definition of policies in XML for resources that may also be expressed in XML. There have also been access control languages that are designed in a logic-based framework (Li et al., 2003; Li and Mitchell, 2003; Li et al., 2002). The additional expressive power and formal, verifiable methodology offered by logic-based languages are particularly useful in the context of access control. Finally, access control languages have also been defined in the context of Semantic Web languages (e.g. Rei (Kagal et al., 2003) and KAoS (Tonti et al., 2003)). The Semantic Web languages are based on description logics which are a decidable subset of first order logic, and hence provide benefits that are similar to logic-based languages.
CHAPTER 3
AIS USING PRIVATE CLOUDS – CLOUD-CENTRIC ASSURED INFORMATION SHARING SYSTEMS

In this chapter, we present solutions using private clouds that address the “secure technological infrastructure” challenge, which was outlined earlier as being one of the key issues that hinders the adoption of AIS within organizations. Although a number of AIS implementations have been developed (Thuraisingham et al., 2008; Awad et al., 2010; Rao et al., 2008), none of them are cloud-based and therefore, they lack the scalability, flexibility and efficiency needed for supporting a large user-base utilizing vast quantities of data. In this chapter, we present details of two prototype systems, namely CAISS-X and CAISS, which provide cloud-based AIS capabilities. The rest of the chapter is organized as follows: In Section 3.1, we outline the need for cloud-based AIS systems. This section also presents the novel contributions in CAISS-X and CAISS. After that, in Sections 3.2 and 3.3 we give a detailed description of the architectures employed by the two prototypes.

3.1 Motivation

There is a tremendous need for the development of efficient and scalable AIS tools and technologies for organizations. The exigence of this need was recently expressed in an NSA document (Hoover, 2011) that outlines the agency’s focus on using a “cloud-centric” approach in the agency’s effort towards improving its information sharing capabilities with other agencies. Similarly, the DoD has endorsed the use of cloud computing to meet its vision of “delivering the power of information to ensure mission success through an agile enterprise with freedom of maneuverability across the information environment” (US-DoD...
Furthermore, the adoption of cloud computing by various big businesses has validated its possible advantages of efficiency, flexibility, scalability and cost-effectiveness. Additionally, the Software as a Service (SaaS) model of cloud computing has made it feasible to conveniently provide AIS as a cloud-based service. Finally, even though a number of policy-based information sharing tools have been developed (Rao et al., 2008; Thuraisingham et al., 2008; Finin et al., 2009), none of these operate in a cloud environment, thus being unable to address the core requirement of large scale data analysis by a significant number of users. This deficiency necessitates the urgent development of various cloud-based AIS systems; preliminary steps in that direction have been made through the implementation of two prototype systems that are described in subsequent sections.

The first of these prototypes, called Cloud-centric Assured Information Sharing System with XACML Policies (CAISS-X), uses a cloud-based framework for managing data and a non-cloud policy engine to enforce XACML policies. CAISS-X uses a layered architecture comprising: (1) A Web Application layer that allows authorized users to share and analyze relational data. (2) An Access Control layer that allows one to build and enforce XACML policies. (3) A Hive layer that serves as a cloud-resident DBMS for managing relational data. (4) A Hadoop HDFS layer that stores data files for tables/views created in Hive.

Although CAISS-X represents a discernible enhancement over prior AIS implementations, it still has the following constraints that are addressed in the second prototype, called Cloud-Centric Assured Information Sharing System (CAISS), which uses a cloud-based framework for both, data storage and retrieval as well as policy enforcement:

- **XACML-based policies**: The types of inter-organizational use cases that can be captured by XACML-based policies is restricted by the limited expressive power of XACML. On the other hand, CAISS uses the power of successively more expressive languages such as RDF and OWL to capture complex use cases that may arise when multiple organizations want to share information.
• **Cloud-based policy engine**: Since CAISS-X uses a non-cloud policy engine, it may not be able to cater to applications involving a large number of users. Conversely, CAISS uses a cloud-resident policy manager to store and enforce policies defined in RDF. Therefore, CAISS achieves the desired goals of scalability, flexibility and efficiency that are integral to mission-critical information sharing tasks.

• **Support for unstructured data**: CAISS-X uses the relational model and therefore requires an *a priori* schema definition, thus making it infeasible for many critical information sharing tasks that operate under rapidly changing mission requirements. Moreover, the relational model is also incapable of representing unstructured data such as text, images, audio and video files. Since CAISS uses the RDF data model, it can easily adapt to changing mission requirements since RDF is schemaless. Furthermore, RDF can also easily represent and process unstructured data.

CAISS uses a layered architecture similar to CAISS-X comprising: (1) A *Web Application* layer that allows authorized users to share and analyze RDF data. (2) A cloud-resident *Access Control* layer that allows one to build and enforce redaction policies, which protect query results released to users by sanitizing sensitive data items in them. (3) A *Jena-HBase* layer that allows users to store and process large RDF graphs in a cloud-based framework. (4) An *HBase* layer that stores tables containing triples that correspond to RDF graphs stored in Jena-HBase. (5) A *Hadoop HDFS* layer that stores data files for tables created in HBase.

### 3.1.1 Our Contributions

CAISS-X and CAISS provide the following:

• Allow secure sharing and analysis of relational/RDF data using Hive/Jena-HBase.

• Provide secure access to data using policy-based mechanisms. CAISS-X uses a non-cloud policy engine that enforces XACML policies, while CAISS uses a cloud-resident
policy engine that enforces redaction policies, thereby providing enhanced expressivity and scalability as compared with CAISS-X.

- Incorporation of the above mechanisms into comprehensive, web-based systems.

3.2 Cloud-centric Assured Information Sharing System with XACML Policies (CAISS-X)

In this section, we present details of CAISS-X, a web-based system that employs a cloud-centric framework to store and query relational data via a non-cloud policy engine that enforces XACML policies (Thuraisingham et al., 2010; Khadilkar et al., 2014). CAISS-X presents a significant improvement over prior non-cloud AIS efforts, and is based on the recommendations of the Cloud Security Alliance for Identity and Access Management (Kumaraswamy et al., 2010) and previous work involving XACML policies (Thuraisingham et al., 2008; Parikh, 2009).

Figure 3.1 presents an architectural overview of CAISS-X, which is broadly divided into the Web Application, Access Control, Hive and HDFS layers (Thuraisingham et al., 2010). Note that, different line styles used in Figure 3.1 denote the flow of control through CAISS-X for various tasks. In particular, the solid line denotes the flow of control for the task of querying data stored in Hive tables/views, the dashed line shows the sequence of steps for creating/loading a new table in Hive and finally, the dotted line represents the procedure followed when a new view needs to be defined in Hive. We now present details of each of the layers in CAISS-X.

**Web Application layer:** This layer provides the only point of entry for users to interact with the cloud infrastructure. CAISS-X currently uses a tight coupling between user

---

1 We acknowledge that preliminary details about the various layers of CAISS-X were presented in (Thuraisingham et al., 2010) and will also be published in (Khadilkar et al., 2014), which is available at: [http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-1-4614-3295-1](http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-1-4614-3295-1) The final publication is available at: [link.springer.com](link.springer.com)
privileges (*viz.* read-only, read/write and administrative) and operations (*viz.* query-only, create/load/query and create/load/query/user-assignment) to define the interaction of a user with the system. In particular, the interaction takes on one of three user roles:

- **read-only, query-only**: A user is only allowed to query pre-existing Hive tables/views.
- **read/write, create/load/query**: Users can create/load tables/views and define XACML policies on them, in addition to being able to query all tables/views.
- **administrative, create/load/query/user-assignment**: This is a specially reserved role for an *admin* user, who in addition to the previously defined functions (create/load/query), can also assign new users to either of the above roles.
Additionally, the web application provides basic authentication using a login page to identify legitimate users. The *usernames* and *passwords* of all users are securely stored in a file using the Java simplified encryption (JASYPT) library’s \cite{Fernández et al. 2010} salted hash technique. Furthermore, this sensitive file is stored in a secure location that is inaccessible to any user.

**Access Control layer**: This layer forms a core component of CAISS-X and is itself composed of several parts, each of which is described below:

- **ZQL Parser**: The ZQL parser \cite{Gibello 2002} validates queries submitted by a user to the Web Application layer. To accomplish this task, ZQL parses a submitted query and if it is unsuccessfully parsed, then an appropriate error message is passed back to the Web Application. On the other hand, if parsing is successful, ZQL fills in various Java Vector\(^2\) objects with results of the parsing process. The Vector objects are then distributed among the different layers of CAISS-X. For example, given the query,

\[
Q = \text{SELECT Student.id, Student.name} \\
\text{FROM Student} \\
\text{WHERE Student.Grade = 'A';},
\]

on successfully parsing \(Q\), ZQL would construct a Vector for each distinct part of \(Q\), namely **SELECT**, **FROM** and **WHERE**. Also, each Vector would be filled with values from \(Q\):

\[
\text{SELECT} = \{\text{Student.id, Student.name}\} \\
\text{FROM} = \{\text{Student}\} \\
\text{WHERE} = \{\text{Student.Grade = 'A'}\}
\]

\(^2\text{http://docs.oracle.com/javase/7/docs/api/index.html} \)
The **SELECT** Vector is then passed back to the Web Application and is used to display column headers for a table containing results of \( Q \). The **FROM** Vector is passed to the XACML Policy Evaluator to ensure that the user submitting \( Q \) has the necessary permissions to access all tables/views specified in the query. If the user has the appropriate permissions, the query (\( Q \) in this case) is evaluated further, else an error message is returned to the Web Application. Note that, ZQL currently provides parsing support only for queries involving the **SELECT**, **INSERT**, **DELETE** and **UPDATE** keywords.

- **XACML Policy Layer**: The XACML Policy Builder and Evaluator together form the XACML Policy layer that is used to construct and enforce XACML policies. We now describe each of these components in additional detail:

  - **XACML Policy Builder**: In CAISS-X, tables/views defined in Hive are treated as resources that need to be protected using XACML policies. Since a user can define policies on both, tables and views, CAISS-X is automatically able to support a fine level of access control. Furthermore, CAISS-X uses Role Based Access Control ([Ferraiolo and Kuhn 1992](#) [Sandhu et al. 1996](#)) for allowing only authorized users to access tables/views defined in Hive. Towards this end, CAISS-X defines a mapping between permissible types of queries and users allowed to run those types of queries. A fictitious sample of such a mapping would be as follows:

    ```
    SELECT root userA userB
    INSERT root userA userC
    ```

    As should be evident, **root** and **userA** have permissions to execute both, **SELECT** and **INSERT** queries, while **userB** can only execute **SELECT** queries and **userC** can only execute **INSERT** queries. When a user wants to create a new table/view, CAISS-X allows them to upload their own XACML policy for that table/view or builds a new policy for them. In the latter case, the user also needs to specify
the types of queries that can be executed on the new table/view. CAISS-X also maintains a separate mapping between table/view name and types of queries that can be executed on that table/view. For example, if userA wants to create a new table table1 and allow SELECT, INSERT and DELETE queries to be executed on table1, the mapping file would be updated with this new information:

```plaintext
table2 SELECT INSERT
table1 SELECT INSERT DELETE
```

Once the mapping file is updated, CAISS-X uses Sun’s XACML implementation (Sun-Microsystems-Inc. 2003) to construct a policy for the new table/view with the associated query types specified by the user.

- **XACML Policy Evaluator**: This component uses the two mappings defined during policy construction and the entire set of XACML policies to ensure that a user is authorized to execute a particular query (viz. SELECT, INSERT, etc.) over the tables/views defined in that query. In practice, CAISS-X uses Sun’s XACML implementation to enforce XACML policies. If a user is authorized to access all tables/views defined in the submitted query, then the query is evaluated further, else an appropriate error message is passed back to the web application. Furthermore, the XACML Policy Evaluator is used both, during query execution as well as during view construction. A view in Hive can only be constructed by executing a SELECT query on existing tables/views. Therefore, in order to create a view, a user must have access to all tables/views that are a part of the view definition’s SELECT query. The policy evaluator is therefore required to ensure that the querying user is authorized to access all the necessary tables/views.

- **Basic Query Rewriting**: This component allows users to submit traditional SQL queries that are rewritten to suit HiveQL’s syntax, thereby adding a layer of abstraction
between users and Hive. CAISS-X currently supports the following simple rewrite rules that can be applied to user submitted SQL queries:

– HiveQL does not permit the use of multiple tables/views in the FROM clause of a query, but rather expects such a query to be defined as a sequence of JOIN statements. CAISS-X allows a user to submit a query with multiple tables in the FROM clause, thereby abstracting a user from HiveQL’s syntax. The rewriting component transforms the submitted query into a new query in which tables in the FROM clause are replaced with a sequence of corresponding JOIN statements. For example, the original SQL query,

\[
Q = \text{SELECT Employee.name, Department.name} \\
\quad \text{FROM Employee, Department;} 
\]

is transformed into the following HiveQL query,

\[
Q' = \text{SELECT Employee.name, Department.name} \\
\quad \text{FROM Employee JOIN Department;} 
\]

– HiveQL supports the following version of SQL’s INSERT-INTO-SELECT statement,

\[
\text{INSERT OVERWRITE TABLE } \text{<tablename>} \\
\text{SELECT <statement>}; 
\]

rather than the traditionally defined,

\[
\text{INSERT INTO } \text{<tablename}> \text{ SELECT <statement>}; 
\]

In this case, CAISS-X allows the traditional definition of the INSERT-INTO-SELECT statement to be used, which is then rewritten to conform to HiveQL’s syntax by simply replacing the keyword INTO with the keywords OVERWRITE TABLE.
**Hive layer:** CAISS-X uses this layer to allow cooperating organizations to share structured data and perform complex analyses on the shared data using HiveQL. A point to observe from Figure 3.1 is that arrows between tables defined in this layer and files stored in the HDFS layer indicate a one-to-one correspondence between tables (e.g. Table1) and files (e.g. File1). Furthermore, a lack of arrows between views defined in this layer and files stored in the HDFS layer suggests that a view is simply a logical concept in Hive that is defined as such using a SELECT query.

**The HDFS layer:** In CAISS-X, this layer is used to store structured data files that correspond to tables created in Hive. The security assumption made in CAISS-X is that users cannot gain access to HDFS files, neither through Hadoop’s command line tool nor through Hadoop’s web interface, and the only way to access files is through the Web Application.

### 3.3 Cloud-centric Assured Information Sharing System (CAISS)

In this section, we present details of CAISS, which employs a cloud-centric framework to store and query RDF graphs via a cloud-resident policy engine that enforces redaction policies.

Figure 3.2 presents an overview of the architecture employed by CAISS. The architecture is broadly divided into the Web Application, Access Control, Jena-HBase, Hive and HDFS layers. Note that, different line styles used in Figure 3.2 denote the flow of control through CAISS for various tasks supported by the system. In particular, the solid line denotes the flow of control in CAISS for the task of querying graphs stored in Jena-HBase, while the dashed line shows the sequence of steps in CAISS for creating/loading a new graph in Jena-HBase. We now present details of each of the layers in CAISS.

**Web Application layer:** This layer uses the exact same code that was developed as a part of the Web Application Layer for CAISS-X. Recall that, this layer provides the only point of entry for users to interact with the cloud infrastructure. Furthermore, it tightly couples
user privileges (viz. read-only, read/write and administrative) and operations (viz. query-only, create/load/query and create/load/query/user-assignment) to define the interaction of a user with the system. Finally, this layer provides a basic authentication mechanism in the form of a login page to identify legitimate users.

**Access Control layer:** This layer forms a core component of CAISS and is composed of a *Policy Builder* and *Evaluator*, which are described below:

- **Policy Builder:** In CAISS, information to be released to a user is protected by sanitizing (equivalently redacting) sensitive data items in query results, which comprise the original user-submitted RDF graphs or their subgraphs produced by executing queries on original graphs. CAISS uses a redaction policy to specify sensitive data
items, which could be a node, an edge connecting two nodes, or a path containing several nodes connected by edges. Since one can sanitize sensitive nodes, edges or paths, CAISS automatically supports fine-grained access control. In practice, CAISS uses the REDACT framework [Rachapalli et al., 2013] to perform sanitization, which contains operations for sanitizing nodes (SNode), edges (SEdge) and paths (SPath).

A redaction policy is defined as an RDF graph and is stored in a special-purpose graph, “Policy Graph”, in Jena-HBase. An example of a fictitious policy could be as follows:

```rdfs
@prefix caiss: <http://cs.utdallas.edu/semanticweb/CAISS/vocab/1.0#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<#policy> rdf:type caiss:Policy ;
   caiss:name "Redact-1" ;
   caiss:policyURI <http://example.org/Data#Dft-Graph-P1> ;
   caiss:resourceURI <http://example.org/Data#Dft-Graph> ;
   caiss:policyDefinition <#defn> ;
.

<#defn> rdf:type caiss:PolicyDefinition ;
   caiss:policyType "Redaction" ;
   caiss:policyOp "SNode ( ?s <http://xmlns.com/foaf/0.1/name> ?o )" ;
.
```

As should be evident, the policy contains the following: (1) A policy name – Redact-1. (2) A unique policy identifier – <http://example.org/Data#Dft-Graph-P1>. (3) The graph to which the policy is applicable – <http://example.org/Data#Dft-Graph>. (4) The type of policy – Redaction. (5) The subgraph that is to be sanitized is defined as – SNode ( ?s <http://xmlns.com/foaf/0.1/name> ?o ), which implies that all “names” in the original graph need to be sanitized.
When a user wants to load a new RDF graph, CAISS allows them to also upload their own predefined policies for that graph or builds a policy for them. In the latter case, the user needs to provide values similar to those specified in the sample policy above. The contents of a policy are stored as triples in the special-purpose “Policy Graph”.

- **Policy Evaluator**: This component receives as input, the results of a user-submitted query and the set of policies stored in “Policy Graph”. Next, the evaluator checks whether sanitization operations defined in any of the policies are applicable to current results. An applicable policy is identified based on the graph being accessed by the user-submitted query. If an applicable policy is identified, the evaluator uses the REDACT framework to apply the sanitization operation in the policy to the results. Finally, the sanitized results are passed on to the web application.

**The Jena-HBase layer**: This layer is used to store user-submitted RDF graphs in a cloud-based framework. RDF provides a cohesive foundation for agile data integration, supports the evolution of applications as business requirements change, and delivers to knowledge workers a greater understanding of the information they possess (Feigenbaum, 2012). The use of a cloud-based framework provides advantages such as scalability, reliability, fault-tolerance, etc. An extensive summary of Jena-HBase is given in Chapter 6.

**The HBase layer**: In CAISS, this layer stores structured tables that correspond to RDF graphs created in Jena-HBase. The security assumption made in CAISS ensures that users cannot gain access to tables, neither through HBase’s command line interface nor through HBase’s web interface, and the only way to access files is through the web application.

**The HDFS layer**: This layer stores data files that correspond to tables created in the HBase layer. Similar to CAISS-X, the security assumption in CAISS is that users cannot gain access to HDFS files, neither through Hadoop’s command line tool nor through Hadoop’s web interface, and the only way to access files is through the web application.
CHAPTER 4
AIS USING HYBRID CLOUDS – HYBRIDIZER: A FRAMEWORK FOR PARTITIONING WORKLOADS OVER HYBRID CLOUDS

The emergence of hybrid clouds has facilitated the development of AIS solutions that allow participating organizations to fully utilize their in-house IT infrastructure for addressing organization-specific requirements, while exploiting low-cost public cloud services for securely sharing data. Before an AIS solution over hybrid clouds can be developed, one needs to have a mechanism that automatically partitions an organization’s data and processing tasks over a hybrid cloud, while taking into account organizational performance, security and financial requirements. In this chapter, we present Hybridizer, a framework that addresses this challenge for various hybrid cloud deployment models, while ensuring that an end-user’s requirements are fulfilled. Hybridizer allows one to fully exploit the benefits of a hybrid cloud, while achieving the right mix of performance, security and financial costs. The rest of the chapter is organized as follows: In Section 4.2 we outline the criteria that drive the design of an effective hybrid cloud solution. Then, in Section 4.3 we formalize the problem of automatically partitioning an organization’s data and processing tasks over a variety of hybrid cloud deployment models, and subsequently, in Section 4.4 we present algorithmic solutions to the problem. Next, Section 4.5 presents an architectural overview of Hybridizer that implements solutions from Section 4.4 that allow automatic partitioning of an organization’s data and processing tasks over a hybrid cloud under user-defined requirements. After that, Section 4.6 presents an overview of a sample problem domain that operates over hybrid clouds, and serves as a testbed for conducting an experimental evaluation of Hybridizer. Finally, Section 4.7 details our extensive experimental investigation into Hybridizer.
4.1 Motivation

The advent of cloud computing has allowed organizations to streamline their operations by focusing on projects with a greater impact on the business, while turning over low-value activities, such as the deployment of IT resources, to a cloud vendor. A hybrid cloud is a particular deployment model composed of two or more cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability [NIST 2011]. Hybrid clouds are well-suited for organizations that place a premium on data security, yet want to achieve dynamic optimizations with respect to operational costs and performance.

Hybrid clouds are particularly useful under three scenarios. First, in an Outsourcing scenario, where an organization relies on a public cloud to fulfill their IT requirements, and uses their limited private cloud to perform supplementary tasks such as filtering incorrect results. Second, in a Cloudbursting setting, where an organization uses their private cloud to develop/deploy applications, and depends on a public cloud to mitigate sudden spikes of activity that arise due to unforeseen circumstances. Third, in a Hybrid scenario, where security-conscious organizations use their private cloud to support mission-critical tasks, and a public cloud to deploy routine tasks. Furthermore, an organization needs to consider several criteria when designing a hybrid cloud application. The most significant criteria include, Performance – an organization would adopt hybrid clouds only if they consistently match their evolving performance requirements, Data Disclosure Risk – the risk an organization takes by storing/processing sensitive information on a public cloud, and Resource Allocation Cost – an organization’s financial expenses arising from their usage of public cloud services.

AIS is a paradigm that allows organizations to securely share information leading to timely decision making without compromising the privacy of individuals [US-DoD 2007a]. An AIS system that integrates critical information collected by government agencies can serve as a potent tool for effectively combating terror. The integrated information can be
analyzed to make proactive decisions that could help to save thousands of lives. A hybrid cloud provides an elegant solution for the above scenario, since each agency can use their in-house IT resources for addressing agency-specific requirements, while a public cloud could be used for secure data integration. However, currently there does not exist a framework that automatically partitions an organization’s data and processing tasks over a hybrid cloud, while taking into account various deployment models and design criteria.

We begin by identifying the criteria that drive the design of a hybrid cloud application, and we also tabulate the applicability of the criteria to various cloud deployment models. A user must make a careful selection of the ingredients, viz. deployment models and design criteria, that they want to use to develop/deploy their applications, while ensuring that their performance, security and financial requirements are fulfilled. Although a user needs to consider a variety of ingredients, the common denominator among any applications developed using these ingredients is that they partition an application’s tasks over a hybrid cloud. In this chapter, we present, Hybridizer, a framework whose goal is to automatically partition an application’s task workload, and in turn, its data, over a hybrid cloud. To achieve this goal, Hybridizer formalizes and solves the problem of automatically dividing an application’s tasks over a hybrid cloud, in the presence of multiple cloud deployment models, such that a user’s mutually conflicting requirements are fulfilled.

4.1.1 Our Contributions

Hybridizer provides the following:

- We identify the design criteria that have the largest impact on an effective hybrid cloud solution. In addition, we tabulate the applicability of these criteria to various hybrid cloud deployment models.

---

1We acknowledge that some of the preliminary details about Hybridizer were presented in (Khadilkar et al., 2012; Oktay et al., 2012), which are available at: ftp://ftp.research.microsoft.com/pub/debull/A12dec/issue1.htm and http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6253510.
• We formalize and solve the problem of automatically distributing an application’s task workload over various hybrid cloud deployment models. Our formalization allows us to test various models for the criteria that underlie a given deployment model. In addition, our formalization provides flexibility by allowing an end-user to test various levels of usage restrictions based on their performance, security and financial requirements.

• We demonstrate the robustness of Hybridizer by conducting an extensive experimental evaluation on a realistic problem domain under a variety of problem parameter settings.

4.2 Design Criteria for Hybrid Cloud Models

There are several key design criteria that contribute towards an effective hybrid cloud solution. In this section, we present a brief overview of each of these criterion. Additionally, in Table 4.1 we show how the design criteria are applicable to different hybrid cloud models (viz. Outsourcing, Cloudbursting and Hybrid) as well as a Private-only cloud.

• Performance: This criterion serves as the main determinant in an end-user’s adoption of hybrid clouds, since one would be motivated to use the hybrid cloud technology only if it meets their evolving performance requirements. Given the abundance of hybrid cloud deployment models, there are several metrics that could be used to measure performance. These include, query response time and network throughput, among others. The performance of a hybrid cloud application is dependent on several criteria such as data model, sensitivity model, etc., which are described below.

• Data Disclosure Risk: Although a user would apply some form of encryption to sensitive data stored on a public cloud, the encrypted data could still leak potentially sensitive information. This factor measures the risk of exposing sensitive information to a service provider [Accenture 2011]. The data disclosure risk depends on the selection
<table>
<thead>
<tr>
<th>Design Criteria</th>
<th>Private-only</th>
<th>Outsourcing</th>
<th>Cloudbursting</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance – Query Response Time, Load balancing, Network Throughput, etc.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data Disclosure Risk – Tuples exposed (Oktay et al., 2012), Analytical model (Fouad et al., 2008), Entropy-based (Trabelsi et al., 2009), etc.</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Resource Allocation Cost – On-premise or Cloud</td>
<td>On-premise</td>
<td>Cloud</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Data Model – Structured, Semi-Structured, Unstructured</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data Partitioning Model – Horizontal, Vertical, Hybrid</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data Replication Model – Full, Partial, None</td>
<td>×</td>
<td>Partial</td>
<td>Full/Partial</td>
<td>Any</td>
</tr>
<tr>
<td>Workload Model – MapReduce jobs, Hive queries in batch form, etc.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sensitivity Model – Privacy associations (Aggarwal et al., 2005), View-based, Predicate-based (Chaudhuri et al., 2007) Miklau and Suciu (2007), etc.</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
made by a user for the sensitivity model, which is outlined below. Additionally, a variety of techniques (Oktay et al., 2012; Fouad et al., 2008; Trabelsi et al., 2009) could be used to model data disclosure risk.

- **Resource Allocation Cost**: This touchstone estimates the financial cost (in $) engendered by a hybrid cloud application’s usage of storage and processing services offered by a public cloud service provider. In general, service providers offer either an elastic pricing model, which charges a user only for the services they use, or a subscription-based pricing model, which charges a user a predetermined fee periodically. Additionally, one could also include the monetary cost of acquiring and maintaining a private cloud, as a part of modeling the problem for a particular hybrid cloud deployment model. Finally, the financial cost estimate of a user’s hybrid cloud application is contingent on the selections made by the user for all the design criteria specified below.

- **Data Model**: This criterion acts as the key to deciding the cloud tools used to develop an application. Consequently, the choice of technology has a direct impact on other criteria such as performance, sensitive data disclosure risks and monetary costs.

- **Data Partitioning Model**: This factor is especially important for hybrid cloud models (particularly, Cloudbursting and Hybrid), since it allows the distribution of data, and thereby computation, over the public and private components of a hybrid cloud. An intelligent partitioning of data over a hybrid cloud can lead to substantial savings for an end-user in terms of performance, data exposure risks and monetary costs.

- **Data Replication Model**: This design choice forms an inseparable component of the hybrid cloud paradigm, since it allows end-users to efficiently distribute computations over a hybrid cloud by maintaining replicas of data on a public cloud. Furthermore, data replication permits users to maintain acceptable levels of data disclosure risk and
resource allocation costs. In this respect, this criterion supplements the previous one in enabling users to satisfy their application requirements.

- **Workload Model**: This criterion complements the Data Model by defining a workload of operations used by an application to access data. The workload model is contingent on a user’s selection of cloud tools, which are subsequently used to construct an application over a hybrid cloud. For example, if an end-user adopts the relational model, the workload could be modeled as a set of Hive queries in batch form. Finally, a user must make a careful selection of a workload model, since it has a direct impact on the performance, security and monetary costs of their hybrid cloud applications.

- **Sensitivity Model**: This touchstone determines the technique used to define what data is sensitive. This criterion is therefore notably important for organizations that deal with sensitive data, which if misused, can lead to severe compliance and litigation expenses. The use of an appropriate sensitivity model provides users with a fine-grained and adaptive strategy for controlling how sensitive data is defined.

**Observations**: There are several observations to be made from the criteria listed above.

- Since the various criteria given above are closely dependent on one another, a methodical selection process is required to successfully preserve the performance, security and financial requirements of an end-user’s application.

- The Data Disclosure Risk serves as the distinguishing touchstone between the Outsourcing vs. the Cloudbursting and Hybrid models. The disclosure risk reaches its highest value in the Outsourcing model, since most of the computations/dataset are outsourced to a public cloud service provider. On the other hand, the disclosure risk
is lower in the Cloudbursting and Hybrid models, since computations and associated data are distributed between the public and private clouds.

- A possible inference that could be made from Table 4.1 is that there is no significant difference between the Cloudbursting and Hybrid models. However, data on a private cloud is always replicated to a public cloud in the Cloudbursting model. The level of replication, *viz.* partial or full, depends on an end-user’s choice. On the other hand, an end-user decides the level of data replication to a public cloud in the Hybrid model.

- A user needs to give the same degree of consideration to each of the criterion we have presented above. However, the following criteria are the most relevant from the perspective of formalizing the workload distribution problem: (1) *Performance*, (2) *Data Disclosure Risk*, and (3) *Resource Allocation Cost*. This is because each of these encapsulate some or all of the remaining criteria. Additionally, we observe that the various criteria mutually conflict with one another.

### 4.3 Workload Distribution Problem (*WDP*)

An organization can effectively use hybrid clouds to construct a scalable, secure and cost-efficient solution to the workload distribution problem. However, there a variety of deployment models that could serve as the foundation of such a hybrid cloud solution. Moreover, as can be seen from Section 4.2 there are a multitude of criteria that affect the design of a solution within a particular hybrid cloud deployment model. Furthermore, each criterion itself presents several options, one or more of which need to be selected by an end-user. Given this large number of choices that need to be made by an end-user, in this section, we present our formulation of the workload distribution problem for various hybrid cloud deployment models. Before we describe our formulation, we introduce some notations in Table 4.2 that are used in the rest of the chapter.
Table 4.2. Notations used within the Task Partitioning Problem

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>The dataset to be partitioned over a hybrid cloud. $R_{pub} \subseteq R$ represents the dataset that is stored on a public cloud.</td>
</tr>
<tr>
<td>$W = {t_1, t_2, \ldots, t_n}$</td>
<td>The workload of tasks to be partitioned across a hybrid cloud. $W_{pub} \subseteq W$ represents the workload that will be executed on a public cloud.</td>
</tr>
<tr>
<td>$Performance(W, W_{pub})$</td>
<td>The performance of workload $W$, given that workload $W_{pub}$ is processed by a public cloud. Note that $W_{pub} \subseteq W$, otherwise it is undefined.</td>
</tr>
<tr>
<td>$Risk(R_{pub})$</td>
<td>The risk of storing data from $R_{pub}$ on a public cloud.</td>
</tr>
<tr>
<td>$Pricing(R_{pub}, W_{pub})$</td>
<td>The monetary cost of using public cloud services for storing data from $R_{pub}$ and processing the workload $W_{pub}$. Note that, $R_{pub} \subseteq R$ and $W_{pub} \subseteq W$.</td>
</tr>
<tr>
<td>$baseData(t)$</td>
<td>The minimum set of data items required to solve a task $t \in W$. We also observe that $\forall m \text{ s.t. } 1 \leq m \leq n; baseData(t_m) \subseteq R$.</td>
</tr>
</tbody>
</table>

Then, we present formalizations of metrics identified earlier as being the most important for hybrid clouds. This is followed by a specification of the workload distribution problem ($WDP$) for each hybrid cloud deployment model.

4.3.1 Metrics used in $WDP$

In this subsection, we provide a description of each of the metrics used within the various formalizations of $WDP$. In particular, each metric is instantiated with a concrete mathematical function that captures high-level aspects of the metric that are relevant to the hybrid cloud setting. This high-level instantiation needs to be further refined for a particular hybrid cloud variant based on the values specified for the Data and Workload models. As an example, we present specific instantiations of the metrics, when the Data and Workload models are based on RDF and SPARQL queries in batch form.
Performance: As stated earlier, an improvement in performance is a primary motivating factor for a user’s adoption of hybrid clouds. Therefore, we use the overall task execution time, $O\text{Run}T(W, W_{pub})$, as an indicator of performance for WDP variants. The overall execution time of tasks in $W$ over a hybrid cloud, given that tasks in $W_{pub}$ are solved on a public cloud can be represented as follows:

$$O\text{Run}T(W, W_{pub}) = \max \left\{ \sum_{t \in W_{pub}} \text{run}T_{pub}(t) \sum_{t \in W - W_{pub}} \text{run}T_{priv}(t) \right\}$$ (4.1)

Note that, $W_{pub} \subseteq W$, otherwise it is undefined. Additionally, $\text{run}T_x(t)$ denotes the estimated running time of task $t \in W$ at site $x$ where $x$ is either a public ($x = pub$) or private ($x = priv$) cloud. In practice, the estimation process depends on several factors such as the individual configuration of a machine in the cloud and the selected Data and Workload models. Since it is difficult to gauge the impact of such factors on processing time, we use the I/O size of a task $t$ as a replacement for the task’s processing time. An I/O-based estimation approach has been previously used to evaluate algorithms for performing query execution on a compute cluster ([Afrati et al., 2011]) and to assess the performance of query plans in the MapReduce framework ([Wu et al., 2011]). The time required to process a task $t$ can be estimated as follows:

$$\text{run}T_x(t) = \sum_{\forall \text{operator } \rho \in t} \frac{\text{inpSize}(\rho) + \text{outSize}(\rho)}{w_t},$$ (4.2)

where $\text{inpSize}(\rho)$ and $\text{outSize}(\rho)$ denote the estimated input and output sizes of an operator $\rho \in t$. Additionally, $w_t$ denotes the no. of I/O operations that can be performed per unit time at site $x$. Note that, $\text{inpSize}(\rho)$ and $\text{outSize}(\rho)$ for an operator $\rho$ can be computed using statistics accumulated over $R$.

When the Data/Workload models are based on RDF/SPARQL queries, $W$ can be modeled as a set of queries $Q$. Then, the execution time of queries in $Q$ over a hybrid cloud,
given that queries in $Q_{pub}$ are executed on a public cloud can be estimated as follows:

$$ ORunT(W, W_{pub}) = ORunT(Q, Q_{pub}) = \max \left\{ \sum_{q \in Q_{pub}} freq(q) \times runT_{pub}(q) \right\} \quad (4.3) $$

where, $freq(q)$ denotes the access frequency of $q \in Q$. The running time of $q$ can be estimated based on the selected query plan $T$ for site $x$ as follows:

$$ runT_x(q) = runT_x(T) = \sum_{\forall \text{operator } \rho \in T} \frac{inpSize(\rho) + outSize(\rho)}{wt_x}. \quad (4.4) $$

**Data Disclosure Risk:** Any event that exposes sensitive information from a public cloud could be disastrous for an organization in terms of financial losses and potential embarrassment. Therefore, an organization would necessarily want to negate such consequences by encrypting sensitive information stored on a public cloud using an appropriate representation. However, various data representations offer varying levels of information protection, for example, plaintext offers no data protection, a bucketized representation \cite{Hacigünumuş et al. 2002} offers more data protection and a random encryption offers the highest level of data protection. Consequently, we adopt a risk-based approach towards safeguarding sensitive information on a public cloud. This notion can be formalized as follows:

$$ Risk(R_{pub}) = \sum_{R_i \in R_{pub}} risk(R_i), \quad (4.5) $$

where $R_{pub}$ is the partition of $R$ that will be stored on a public cloud. On the other hand, $risk(R_i)$ computes the data disclosure risk associated with storing a data item (e.g. an attribute of a relation, a Key-Value pair, etc.), $R_i \in R_{pub}$, on a public cloud. Note that, quantifying risk in terms of a particular data representation is a nontrivial task, and therefore, we leave such an undertaking for the future. When $R$ is based on RDF, one could use the total number of sensitive nodes in a data item $R_i \in R_{pub}$ as a measure of disclosure risk.
This can be formulated as: \( \text{risk}(R_i) = \text{sens}(R_i) \), where \( \text{sens}(R_i) \) is the number of sensitive nodes in \( R_i \). In this instance, \( R_i \) could be a subgraph of a graph, a predicate of a graph, etc.

**Resource Allocation Cost:** Since public cloud services are used on an ongoing basis, they are usually included in the operational expenditure (OpEx) of an organization. Therefore, a strategic way to curtail this expenditure could be to possibly restrict the data and processing outsourced to a public cloud. We estimate the financial cost of utilizing public cloud services as follows:

\[
\text{Pricing}(R_{pub}, W_{pub}) = \text{store}(R_{pub}) + \sum_{t \in W_{pub}} \text{proc}(t) \tag{4.6}
\]

where \( \text{store}(R_{pub}) \) represents the monetary cost associated with storing a subset \( R_{pub} \subseteq R \) on a public cloud while \( \text{proc}(t) \) denotes the monetary cost associated with processing a task \( t \in W_{pub} \) on a public cloud. Note that, \( \text{proc}(t) \) can be further refined based on the Workload model used in a given problem setting. For example, if the Workload model is based on SPARQL queries, \( W \) can be modeled as a set of queries \( Q \). Then, the cost of processing a query \( q \in Q_{pub} \), can be estimated as follows:

\[
\text{proc}(t) = \text{freq}(q) \times \text{proc}(q), \tag{4.7}
\]

where \( \text{freq}(q) \) denotes the access frequency of \( q \in Q \). Additionally, per unit data storage and processing costs are dependent on the particular cloud provider selected by a user.

### 4.3.2 WDP Variants for Hybrid Cloud Models

In this section, we present variants of WDP tailored to the requirements of each of the hybrid cloud models, namely Outsourcing, Cloudbursting and Hybrid.

**WDP Definition for Outsourcing Model:** The cloud computing paradigm has made it feasible for service providers to offer hardware and software solutions over the Internet. An organization could capitalize on this situation by offloading a majority of their data
and processing tasks to such service providers. In addition, an organization could use their limited private cloud infrastructure to store/process critical tasks. This usage model provides organizations with benefits such as a reduction in overall costs, access to newer technology and a reduction in liability. Given dataset $R$, task workload $W$ and the deployment model just described, an Outsourcing variant of $WDP$ could be modeled as an optimization problem whose goal is to find subsets $W_{pub} \subseteq W$ and $R_{pub} \subseteq R$ such that the overall data disclosure risk is minimized. The objective function aims to reduce sensitive data exposure to a public cloud by maximizing the use of the private cloud to store/process sensitive information.

$$\text{minimize} \quad \sum_{R_i \in R_{pub}} \text{risk}(R_i)$$

subject to

1. $O\text{RunT}(W, W_{pub}) \leq \text{DEADLINE\_CONST}$

2. $\text{store}(R_{pub}) + \sum_{t \in W_{pub}} \text{proc}(t) \leq \text{PRA\_CONST}$

3. $\forall t \in W_{pub}, \text{baseData}(t) \subseteq R_{pub}$

where $\text{DEADLINE\_CONST}$ and $\text{PRA\_CONST}$ denote the maximum permissible workload performance deadline and resource allocation cost. The constraints specified above capture the following aspects of the Outsourcing model: (1) Constraint (1) ensures that the overall performance of the workload is better than a user-specified deadline value. (2) Constraint (2) sets an upper limit on the amount of public cloud services that can be obtained through a user-specified monetary value. (3) Constraint (3) ensures that data items required to solve all tasks in $W_{pub}$ are stored on a public cloud.

$WDP$ Definition for Cloudbursting Model: The private cloud infrastructure deployed by an organization could be insufficient during a scenario in which user generated traffic experiences a spike during peak demand, for example, during a holiday season or a widely advertised sales event. An organization usually deals with such a scenario by provisioning public cloud services to ensure that the overall demand is satisfied. This usage
model provides organizations with several advantages such as flexibility, fault tolerance and a smaller operational expense. Given dataset \( R \), task workload \( W \) and the deployment model just described, a Cloudbursting variant of \( WDP \) could be modeled as an optimization problem whose goal is to find subsets \( W_{pub} \subseteq W \) and \( R_{pub} \subseteq R \) such that the public cloud resource allocation cost is minimized. In this case, the goal of the objective function is to reduce the usage of public cloud services, while exposing as little sensitive data as possible and ensuring that all tasks are completed within the specified performance deadline.

\[
\begin{align*}
\text{minimize} & \quad store(R_{pub}) + \sum_{t \in W_{pub}} \text{proc}(t) \\
\text{subject to} & \quad (1) \ ORunT(W, W_{pub}) \leq \text{DEADLINE\_CONST} \\
& \quad (2) \ \sum_{R_i \in R_{pub}} \text{risk}(R_i) \leq \text{DISC\_CONST} \\
& \quad (3) \ \forall t \in W_{pub}, \text{baseData}(t) \subseteq R_{pub}
\end{align*}
\]

where \( \text{DISC\_CONST} \) and \( \text{DEADLINE\_CONST} \) denote the maximum permissible data disclosure risk and workload performance deadline as specified by a user. The constraints specified above capture the following aspects of the Cloudbursting model: (1) Constraint (1) ensures that only a user-specified amount of data is exposed to a public cloud. (2) Constraint (2) ensures that the overall performance of the workload is better than a user-specified deadline value. (3) Constraint (3) ensures that data items required to solve all tasks in \( W_{pub} \) are stored on a public cloud.

**WDP Definition for Hybrid Model:** In certain situations, an organization may be best served by performing an \textit{a priori} distribution of the workload over public and private clouds. For example, an organization could always schedule mission-critical tasks that operate over sensitive data on a private cloud. At the same time, tasks that are data, memory or processor intensive and need to be performed on a regular basis could be scheduled on a public cloud. Such a deployment model offers an organization the right mixture of data
security, level of performance and financial costs. Given dataset $R$, task workload $W$ and the deployment model just described, a Hybrid variant of $WDP$ could be modeled as an optimization problem whose goal is to find subsets $W_{pub} \subseteq W$ and $R_{pub} \subseteq R$ such that the overall execution time of $W$ is minimized. In this case, the purpose of the objective function is to distribute a workload over public and private clouds such that an optimum level of query performance is achieved.

$$
\text{minimize} \quad ORunT(W, W_{pub}) \\
\text{subject to} \quad (1) \quad \sum_{R_i \in R_{pub}} \text{risk}(R_i) \leq \text{DISC\_CONST} \\
(2) \quad \text{store}(R_{pub}) + \sum_{t \in W_{pub}} \text{proc}(t) \leq \text{PRA\_CONST} \\
(3) \forall t \in W_{pub}, \text{baseData}(t) \subseteq R_{pub}
$$

where, as before, $\text{DISC\_CONST}$ and $\text{PRA\_CONST}$ denote the maximum permissible data disclosure risk and public cloud resource allocation cost as specified by an end-user. Additionally, constraints 1 – 3 specified above retain the same semantics as described earlier.

4.4 Algorithmic Solutions to the Workload Distribution Problem

The goal of algorithmic solutions to $WDP$ should be to conduct an efficient search over the solution space so as to find an optimum division of dataset $R$ and workload $W$ over a hybrid cloud. Also, a solution to a $WDP$ variant is only required to find a subset $W_{pub} \subseteq W$ of the task workload, since the corresponding subset $R_{pub} \subseteq R$ of the dataset is equivalent to $(\bigcup_{t \in W_{pub}} \text{baseData}(t))$. A different solution that contains the subset, $W' \subseteq W : W' \neq W_{pub}$, will include a subset $R' \subseteq R : R' \neq R_{pub} \land R' \supseteq R_{pub}$ of the dataset. However, the original solution, $(W_{pub}, R_{pub})$, preserves constraints better than the new solution, $(W', R')$. Finally, since we have three variants of $WDP$ with differing objective functions, we subsequently present three different algorithmic solutions, one for each variant.
Algorithm 1 \textsc{Solve-WDP-Outsourcing()}

\textbf{Input:} $W$, \textit{DEADLINE\_CONST}, \textit{PRA\_CONST}

\textbf{Output:} $W_{\text{pub}}$

1: Initialize $pubW[][][]$
2: for $w = 1 \rightarrow W.$size do
3: \hspace{1em} \textit{time} $\leftarrow$ \textit{ORunT}(\textit{t}w, \emptyset)
4: \hspace{1em} \textit{procCost} $\leftarrow$ \textit{proc}(\textit{t}w); \textit{monCost} $\leftarrow$ \text{store}(\textit{baseData}(\textit{t}w)) $+$ \textit{procCost}
5: for $x = 1 \rightarrow \text{DEADLINE\_CONST} + 1$ do
6: \hspace{2em} for $y = 1 \rightarrow \text{PRA\_CONST} + 1$ do
7: \hspace{3em} if $x < \textit{time}$ then $\{pubW[w-1][x-1][y-1] \leftarrow \{NaN\}; \text{continue;\}}$
8: \hspace{3em} if $w == 1$ then
9: \hspace{4em} if $\textit{risk}(t_1) == 0$ AND $\textit{monCost} \leq y$
10: \hspace{4em} then $pubW[w-1][x-1][y-1] \leftarrow \{t_1\}$
11: \hspace{4em} else $pubW[w-1][x-1][y-1] \leftarrow \emptyset$
12: \hspace{3em} else
13: \hspace{4em} \textit{pubDisc} $\leftarrow$ $\infty$; $(x',y') \leftarrow (\textit{NaN,NaN})$; \textit{priDisc} $\leftarrow$ $\infty$; $(x'') \leftarrow (\textit{NaN})$
14: \hspace{3em} if $\textit{monCost} \leq y$ then
15: \hspace{4em} for all $x - \text{time} \leq iC \leq x; y - \textit{monCost} \leq iD \leq y - \textit{procCost}$ do
16: \hspace{5em} if $pubW[w-1][iC][iD] \neq \textit{NaN}$ then
17: \hspace{6em} $\textit{tmp} \leftarrow pubW[w-1][iC][iD] \cup \{t_w\}$
18: \hspace{6em} if $\textit{constr}($\textit{tmp},\textit{iC, iD}) \text{ AND } \textit{risk}($\textit{tmp}) < \textit{pubDisc}$
19: \hspace{6em} then $\textit{pubDisc} \leftarrow \textit{risk}($\textit{tmp}); $(x',y') \leftarrow (iC,iD)$
20: \hspace{4em} end if
21: \hspace{3em} end for
22: \hspace{3em} end if
23: \hspace{3em} for all $x - \text{time} \leq iC \leq x$ do
24: \hspace{4em} if $pubW[w-1][iC][y] \neq \textit{NaN}$ then
25: \hspace{5em} $\textit{tmp} \leftarrow pubW[w-1][iC][y]$\n26: \hspace{5em} if $\textit{constr}($\textit{tmp},\textit{iC, y}) \text{ AND } \textit{risk}($\textit{tmp}) < \textit{priDisc}$
27: \hspace{5em} then $\textit{priDisc} \leftarrow \textit{risk}($\textit{tmp}); $(x'') \leftarrow (iC)$
28: \hspace{4em} end if
29: \hspace{4em} end for
30: \hspace{3em} if $\textit{priDisc} \leq \textit{pubDisc}$
31: \hspace{4em} then $pubW[w-1][x-1][y-1] \leftarrow pubW[w-1][x''][y-1]$
32: \hspace{4em} else $pubW[w-1][x-1][y-1] \leftarrow pubW[w-1][x'][y] \cup \{t_w\}$
33: \hspace{3em} end if
34: \hspace{3em} end for
35: \hspace{3em} end for
36: \hspace{1em} return $pubW[W.$size $- 1][\textit{DEADLINE\_CONST}][\textit{PRA\_CONST}]$
**Algorithmic Solution to Outsourcing variant of WDP**: Algorithm receives as inputs, a workload $W$ and constraints, namely $DEADLINE\_CONST$ and $PRA\_CONST$. The algorithm produces as its output, the set of tasks from $W$ that will be processed on a public cloud, $W_{pub}$. Algorithm begins by initializing an array, $pubW$, which holds the set of tasks corresponding to $W_{pub}$ (line 1). Next, the algorithm iterates over all tasks in $W$ (lines 2-36). In each iteration, the algorithm determines whether the current task, namely $t_w$, is a part of $W_{pub}$. Then, the algorithm computes values for the following metrics for task $t_w$ (lines 3-4): (1) The approximate time required to execute the task, $time$. (2) The monetary cost of executing $t_w$ on a public cloud, $procCost$. (3) The overall resource allocation cost for processing $t_w$, $monCost$. Next, the algorithm iterates over all possible deadline and resource allocation cost values (up to $DEADLINE\_CONST+1$ and $PRA\_CONST+1$ respectively) to determine whether task $t_w$ is a part of the optimal task set (lines 6-35).

In each iteration, the algorithm first checks if the current deadline value is less than the time required for task $t_w$ (line 7). If it is, then the corresponding location in $pubW$ is set to $NaN$ and the algorithm continues. This check serves as a way to eliminate cases in which the deadline constraint is violated. Next, the algorithm checks if $t_w$ is the first task from $W$ (line 8). If it is, then we check if the disclosure risk associated with $t_1$ is 0 and $monCost$ is less than the current monetary cost value (line 9). If both checks are satisfied, then the corresponding location in $pubW$ is assigned $t_1$ (line 10), else it is left empty (line 11). On the other hand, if the algorithm is processing a task $t_w$ other than $t_1$, then multiple subproblems need to be investigated for the case when $t_w$ is processed on a public cloud (lines 12-33), since this results in an increase in the overall deadline and resource allocation cost. The exact increase depends on the optimal task set, $W_s$, over all previous tasks, $t_1, \ldots, t_{w-1}$, located between positions, $pubW[w-1][0][0]$ and $pubW[w-1][DEADLINE\_CONST][PRA\_CONST]$. Furthermore, the data items required to solve $t_w$, $baseData(t_w)$, could already be partially included in set, $Ws$, constructed thus far. Therefore, storing $baseData(t_w)$ along with $\bigcup_{t \in W_s} baseData(t)$,
may not increase the deadline and resource allocation cost by $time$ and $monCost$, which are the costs of the metrics for $t_w$. Consequently, we need to investigate all positions between $pubW[w - 1][x - time][y - monCost]$ and $pubW[w - 1][x][y - procCost]$ in order to optimally place $t_w$. Moreover, every candidate set of tasks formed by taking the union of $t_w$ with the tasks from $pubW$ should be tested to find out the best solution in terms of disclosure risk that does not violate any constraint (lines 18-19). In case an optimal solution is obtained (line 19), it becomes one of two candidate solutions, with the other solution being when $t_w$ is processed on a private cloud.

To find a candidate solution for the private cloud, we use a procedure similar to the one used for the public cloud. However, since executing a query on a private cloud does not result in any monetary expenditure, we only need to investigate all positions between $pubW[w - 1][x - time][y]$ and $pubW[w - 1][x][y]$ in order to optimally place $t_w$. Moreover, every candidate set of tasks from $pubW$, without task $t_w$, which is the task currently under consideration, should be tested to find out the best solution in terms of disclosure risk that does not violate any constraint (lines 26-27). In case an optimal solution is obtained (line 27), it becomes the candidate solution for the private cloud. After optimal solutions for the public and private clouds are computed, the algorithm compares the disclosure risk of both solutions, and assigns $t_w$ to the cloud with the lower overall risk (lines 30-32). Finally, Algorithm 1 returns the optimal task set containing tasks to be processed on a public cloud, which is located at $pubW[W.size - 1][DEADLINE\_CONST][PRA\_CONST]$ (line 37).

Algorithmic Solution to Cloudbursting variant of WDP: Algorithm 2 receives as inputs, a workload $W$ and constraints, namely $DEADLINE\_CONST$ and $DISC\_CONST$. The algorithm produces as its output, the set of tasks from $W$ that will be processed on a public cloud, $W_{pub}$. Algorithm 2 begins by initializing an array, $pubW$, which holds the set of tasks corresponding to $W_{pub}$ (line 1). Next, the algorithm iterates over all tasks in $W$ (lines 2-36). In each iteration, the algorithm determines whether the current task, namely
Algorithm 2 Solve-WDP-Cloudbursting()

Input: $W$, DEADLINE CONST, DISC CONST
Output: $W_{pub}$

1: Initialize $pubW[][]$
2: for $w = 1 \rightarrow W.size$ do
3:  $\text{time} \leftarrow \text{ORunT}(t_w, \emptyset); \text{discCost} \leftarrow \text{risk}(t_w)$
4: for $x = 1 \rightarrow \text{DEADLINE CONST} + 1$ do
5:   for $y = 1 \rightarrow \text{DISC CONST} + 1$ do
6:     if $x < \text{time}$ then 
7:       $\{pubW[w-1][x-1][y-1] \leftarrow \{\text{NaN}\}; \text{continue};\}$
8:     if $w == 1$ then
9:       if $\text{discCost} \leq y$ then $\{pubW[w-1][x-1][y-1] \leftarrow \{t_1\}$
10:      else $\{pubW[w-1][x-1][y-1] \leftarrow \emptyset$  
11:    else
12:       $\text{pubPRA} \leftarrow \infty; (x', y') \leftarrow (\text{NaN}, \text{NaN}); \text{priPRA} \leftarrow \infty; (x'') \leftarrow (\text{NaN})$
13:      if $\text{discCost} \leq y$ then
14:        for all $x - \text{time} \leq iC \leq x; \text{y - discCost} \leq iD \leq y$ do
15:          if $\text{pubW[w-1][iC][iD]} \neq \text{NaN}$ then
16:            $\text{tmp} \leftarrow \text{pubW[w-1][iC][iD]} \cup \{t_w\}$
17:            $\text{monCost} \leftarrow \text{store}(\text{tmp}) + \sum_{t \in \text{tmp}} \text{proc}(t)$
18:            if constr($\text{tmp}, iC, iD$) AND $\text{monCost} < \text{pubPRA}$
19:              then $\text{pubPRA} \leftarrow \text{monCost}; (x', y') \leftarrow (iC, iD)$
20:        end if
21:      end for
22:    end if
23:  for all $x - \text{time} \leq iC \leq x$ do
24:    if $\text{pubW[w-1][iC][y]} \neq \text{NaN}$ then
25:      $\text{tmp} \leftarrow \text{pubW[w-1][iC][y]}$
26:      $\text{monCost} \leftarrow \text{store}(\text{tmp}) + \sum_{t \in \text{tmp}} \text{proc}(t)$
27:    if constr($\text{tmp}, iC, y)$ AND $\text{monCost} < \text{priPRA}$
28:      then $\text{priPRA} \leftarrow \text{monCost}; (x'') \leftarrow (iC)$
29:    end if
30:  end for
31: if $\text{priPRA} \leq \text{pubPRA}$
32:  then $\text{pubW[w-1][x-1][y-1]} \leftarrow \text{pubW[w-1][x''][y-1]}$
33: else $\text{pubW[w-1][x-1][y-1]} \leftarrow \text{pubW[w-1][x'][y'] \cup \{t_w\} }
34: end if
35: end for
36: end for
37: return $\text{pubW[W.size - 1][DEADLINE CONST][DISC CONST]}$
$t_w$, is a part of $W_{pub}$. Then, the algorithm computes values for the following metrics for task $t_w$ (line 3): (1) The approximate time required to execute the task, $time$. (2) The data disclosure risk, $discCost$. Next, the algorithm iterates over all possible deadline and disclosure risk values (up to $DEADLINE\_CONST+1$ and $DISC\_CONST+1$ respectively) to determine whether task $t_w$ is a part of the optimal task set (lines 4-35).

In each iteration, the algorithm first checks if the current deadline value is less than the time required for task $t_w$ (line 6). If it is, then the corresponding location in $pubW$ is set to $NaN$ and the algorithm continues. This check serves as a way to eliminate cases in which the deadline constraint is violated. Next, the algorithm checks if $t_w$ is the first task from $W$ (line 7). If it is, then we check if the disclosure risk is less than the current disclosure risk value (line 8). If the check is satisfied, then the corresponding location in $pubW$ is assigned $t_1$ (line 8), else it is left empty (line 9). On the other hand, if the algorithm is processing a task $t_w$ other than $t_1$, then multiple subproblems need to be investigated for the case when $t_w$ is processed on a public cloud (lines 10-33), since this results in an increase in the overall deadline and disclosure risk. The exact increase depends on the optimal task set, $W_s$, over all previous tasks, $t_1,\ldots,t_{w-1}$, located between positions, $pubW[w-1][0][0]$ and $pubW[w-1][DEADLINE\_CONST][DISC\_CONST]$. Furthermore, the data items required to solve $t_w$, $baseData(t_w)$, could already be partially included in set, $W_s$, constructed thus far. Therefore, storing $baseData(t_w)$ along with $\bigcup_{t \in W_s} baseData(t)$, may not increase the deadline and disclosure risk by $time$ and $discCost$, which are the costs of the metrics for $t_w$. Consequently, we need to investigate all positions between $pubW[w-1][x-time][y-discCost]$ and $pubW[w-1][x][y]$ in order to optimally place $t_w$. Moreover, every candidate set of tasks formed by taking the union of $t_w$ with the tasks from $pubW$ should be tested to find out the best solution in terms of resource allocation cost that does not violate any constraint (lines 17-18). In case an optimal solution is obtained (line 18), it becomes one of two candidate solutions, with the other solution being when $t_w$ is processed on a private cloud.
To find a candidate solution for the private cloud, we use a procedure similar to the one used for the public cloud. However, since executing a query on a private cloud does not result in any disclosure risk, we only need to investigate all positions between $pub_W[w-1][x-time][y]$ and $pub_W[w-1][x][y]$ in order to optimally place $t_w$. Moreover, every candidate set of tasks from $pub_W$, without task $t_w$, which is the task currently under consideration, should be tested to find out the best solution in terms of resource allocation cost that does not violate any constraint (lines 26-27). In case an optimal solution is obtained (line 27), it becomes the candidate solution for the private cloud. After optimal solutions for the public and private clouds are computed, the algorithm compares the resource allocation costs of both solutions, and assigns $t_w$ to the cloud with the lower overall cost (lines 30-32). Finally, Algorithm 2 returns the optimal task set containing tasks to be processed on a public cloud, which is located at $pub_W[W.size-1][DEADLINE\_CONST][DISC\_CONST]$ (line 37).

**Algorithmic Solution to Hybrid variant of WDP:** Algorithm 3 receives as its inputs, a workload $W$ and constraints, namely $DISC\_CONST$ and $PRA\_CONST$. The algorithm produces as its output, the set of tasks from $W$ that will be processed on a public cloud, $W_{pub}$. Algorithm 3 begins by initializing an array, $pubW$, which holds the set of tasks that will be processed on a public cloud (line 1). Next, the algorithm iterates over all tasks in workload $W$ (lines 2-23). In each iteration, the algorithm determines whether the current task, $t_w$, will be processed on a public cloud. The algorithm first computes values for the following metrics for task $t_w$ (lines 3-4): (1) The data disclosure risk, $discCost$. (2) The monetary cost of executing $t_w$ on a public cloud, $procCost$. (3) The overall resource allocation cost for processing $t_w$ on a public cloud, $monCost$. Next, the algorithm iterates over all possible risk and resource allocation cost values (up to $DISC\_CONST$ and $PRA\_CONST$ respectively) to determine whether task $t_w$ is a part of the optimal task set (lines 5-22).

In each iteration, the algorithm first checks if the task being currently analyzed is the first task from $W$ (line 6). If it is, then the algorithm checks whether $t_1$ violates any constraint
Algorithm 3 SOLVE-WDP-HYBRID()

Input: $W$, DISC_CONST, PRA_CONST
Output: $W_{pub}$

1: Initialize $pubW[][]$
2: for $w = 1 \rightarrow W.size$ do
3:   $discCost \leftarrow \sum_{R_i \in baseData(t_w)} \text{risk}(R_i)$
4:   $procCost \leftarrow \text{proc}(t_w)$; $monCost \leftarrow \text{store}(baseData(t_w)) + \text{procCost}$
5:   for $x = 1 \rightarrow DISC\_CONST + 1$; $y = 1 \rightarrow PRA\_CONST + 1$ do
6:     if $w == 1$ then
7:       if checkConstr($\{t_1\}, x, y$) AND $\text{ORunT}(W^1, W^1) < \text{ORunT}(W^1, \emptyset)$
8:       then $pubW[w-1][x-1][y-1] \leftarrow \{t_1\}$ else $pubW[w-1][x-1][y-1] \leftarrow \emptyset$
9:     else
10:        $pubPerf \leftarrow \infty$; $(x', y') \leftarrow (NaN, NaN)$
11:       if checkConstr($\{t_w\}, x, y$) then
12:          for all $x - discCost \leq iC \leq x$; $y - monCost \leq iD \leq y - procCost$ do
13:             $tmp \leftarrow pubW[w-1][iC][iD] \cup \{t_w\}$
14:             if checkConstr($tmp, iC, iD$) AND $\text{ORunT}(W^w, tmp) < pubPerf$
15:                then $pubPerf \leftarrow \text{ORunT}(W^w, tmp)$; $(x', y') \leftarrow (iC, iD)$
16:          end for
17:       end if
18:       $privPerf \leftarrow \text{ORunT}(W^w, pubW[w-2][x-1][y-1])$
19:       if $privPerf \leq pubPerf$ then $pubW[w-1][x-1][y-1] \leftarrow pubW[w-2][x-1][y-1]$
20:       else $pubW[w-1][x-1][y-1] \leftarrow pubW[w-2][x'[y'] \cup \{t_w\}$
21:     end if
22:   end for
23: end for
24: return $pubW[W.size - 1][DISC\_CONST][PRA\_CONST]$

for the current risk and resource allocation cost, by using method checkConstr($\{t_1\}, x, y$).
Additionally, the algorithm also checks whether processing $t_1$ on a public cloud leads to a better overall performance than processing it on a private cloud (line 7). If none of the constraints are violated and the public cloud produces better results than the private cloud, then the corresponding array location in $pubW$ is assigned task $t_1$ (line 8), else it is left empty (line 8). On the other hand, if the algorithm is processing a task $t_w$ other than task $t_1$, then multiple subproblems need to be investigated for the case when $t_w$ is processed on a public...
cloud (lines 9-21). This is because, processing task $t_w$ on a public cloud results in an increase in the overall risk and resource allocation cost. The exact increase in these values depends on the optimal task set, $W_s$, over all previous tasks, $t_1, \ldots, t_{w-1}$, located between array positions, $pubW[w-1][0][0]$ and $pubW[w-1][DISC\_CONST][PRA\_CONST]$. Furthermore, the data items required to solve $t_w$, $baseData(t_w)$, could already be partially included in the optimal task set, $W_s$, constructed thus far. Therefore, storing $baseData(t_w)$ along with $\bigcup_{t \in W_s} baseData(t)$, may not increase the overall risk and resource allocation cost by $discCost$ and $monCost$, which represent the costs of the metrics for task $t_w$. Consequently, we need to investigate all array positions between $pubW[w-1][x-discCost][y-monCost]$ and $pubW[w-1][x][y-procCost]$ in order to optimally place task $t_w$. Moreover, every candidate set of tasks formed by taking the union of task $t_w$ with the set of tasks from $pubW$ should be tested to find out the best overall solution in terms of performance that also does not violate any constraint (lines 14-15). In case an optimal solution is obtained (line 15), it becomes one of two overall candidate solutions, with the other solution being the case when task $t_w$ is processed on a private cloud (line 18). After the optimal solutions for the public and private clouds are computed, the algorithm compares the performance of both solutions, and assigns task $t_w$ to the cloud with the better overall performance (lines 19-20). Finally, Algorithm 3 returns the optimal task set, located at $pubW[W.size-1][DISC\_CONST][PRA\_CONST]$, that contains tasks that will be processed on a public cloud (line 24).

4.5 Architectural Overview of Hybridizer

The goal of Hybridizer is to automatically distribute data/computations over a public and private cloud in accordance with the requirements of a user’s application. To achieve this goal, Hybridizer’s architecture should incorporate solutions for $WDP$ variants. Furthermore, the architecture should be adaptable to any of the three cloud deployment models. Figure 4.1 presents an architectural overview of Hybridizer that achieves all previous goals. Note that,
once a deployment model has been selected, only the affiliated blocks from Figure 4.1 will be used. For example, if a Hybrid model has been selected, only the Disclosure Risk Estimator and Monetary Cost Estimator will be used by the Workload Distribution Module.

_Hybridizer’s_ architecture consists of the following components: (1) The **Statistics Gathering Layer** accumulates statistics for dataset $R$ and workload $W$. (2) The **Data and Query Management Layer** provides a solution for automatically distributing dataset $R$ and workload $W$ over public and private clouds. (3) The **Cloud Layer** stores dataset $R$ across a hybrid cloud, and coordinates the execution of tasks from $W$ over the distributed data. The following steps capture the flow of information in _Hybridizer’s_ architecture:

1. A user submits the following: (i) The dataset $R$ to be partitioned. (ii) The task workload $W$ to be distributed. (iii) A set of constraints, $C$, which varies depending on
the selected deployment model. $C$ consists of values denoting the maximum permissible data disclosure risk, public cloud resource allocation cost and performance deadline.

2. The dataset $R$ and workload $W$ are analyzed by the statistics gathering layer, which also stores the results of the analysis as equi-width histograms in $SR$. The histograms maintain various statistics such as the average length of a value and the number of values, in a given subset $R_i \in R$. In addition, the statistics gathering layer also estimates the following information: (i) The minimum set of data items required to solve a task $t \in W$ is stored in $baseData(t)$. (ii) The running time of a task $t \in W$ for both public ($x = pub$) and private ($x = priv$) clouds is stored in $runT_x(t)$.

3. The results of step (2) are forwarded to estimator modules. The results include, histograms $SR$ and, $\forall t \in W$, their running times on the public/private clouds, $runT_x(t)$, and the estimated minimum set of data items required to process $t$, $baseData(t)$.

4. The dataset $R$, workload $W$ and constraints $C$ are transmitted from the User Interface Layer to the Workload Distribution Module.

5. The Workload Distribution Module uses inputs received in step (4) to solve the workload distribution problem ($WDP$). In solving $WDP$, this module uses the appropriate estimator modules based on the cloud deployment model selected by an end-user. The Performance Estimator estimates the overall performance of workload $W$ over a hybrid cloud. The Disclosure Risk Estimator calculates the risk associated with storing data on a public cloud. The Monetary Cost Estimator computes the financial cost of storing data and processing tasks on a public cloud.

6. The Workload Distribution Module produces the following outputs for a public cloud (step (6a)): (i) $R_{pub} \subseteq R$ is the subset of $R$ that will be stored on a public cloud. (ii) $W_{pub} \subseteq W$ is the subset of $W$ that will be processed on a public cloud. In addition,
the module also produces subsets $R_{priv} \subseteq R$ and $W_{priv} \subseteq W$ that respectively denote the data stored and tasks processed on a private cloud (step (6b)). Note that, $R$ could be replicated entirely to a public or private cloud depending on the data replication and cloud deployment model selected by a user.

7. The **Storage/Query Module** at each cloud site stores and processes the appropriate subset, $R_{pub}/W_{pub}$ or $R_{priv}/W_{priv}$. The selection of tools/technologies to be used within the Storage/Query Module depends on the Data Model/Workload Model requirements of an end-user application. The results of tasks from $W_{pub}/W_{priv}$ are directly transmitted to a user application (step (7)).

In addition to the architectural overview given above, *Hybridizer* also contains a runtime component, shown in Figure 4.2 which assigns a task $t'$ to either the public or private cloud. This component receives as its inputs, $R_{pub}$ and $Load(Pu)$ from the public cloud (Step (1a)), and $R_{priv}$ and $Load(Pr)$ from the private cloud (Step (1b)), where $Load(Pu)$ and $Load(Pr)$ denote the current estimated computational load on the public and private cloud.
clouds respectively. Note that, the load on a cloud could be measured in a number of ways, including overall response time, total CPU utilization, etc. Next, the component checks whether \( baseData(t') \subseteq R_{priv} \land Load(Pr) < PRIV\_CAP \), i.e., whether the set of data items required to solve \( t' \) are available on the private cloud and if computational resources are available on the private cloud to execute \( t' \). If the conditions are satisfied, the component assigns \( t' \) to the private cloud (step (2)). On the other hand, if either of these conditions is not satisfied, then the component checks if \( t' \) could be assigned to the public cloud using, \( baseData(t') \subseteq R_{pub} \land Load(Pu) < PUB\_CAP \). Again, if the conditions are satisfied, the component assigns \( t' \) to the public cloud (step (2)). In the event that either of these conditions is also not satisfied, the processing of \( t' \) will be delayed until a repartitioning of \( R \) that makes \( baseData(t') \subseteq R_{priv} \lor baseData(t') \subseteq R_{pub} \) is achieved or until the computational load on one of the two sides drops below the maximum capacity.

4.6 Sample Problem Domain over Hybrid Clouds

In this section, we present an overview of a sample problem domain that operates over hybrid clouds, and serves as a testbed for conducting an experimental evaluation of Hybridizer. The sample problem domain is constructed through a combination of a particular cloud deployment model and a specification of values for the various design criteria we outlined earlier. The rationale behind the development of the problem domain was to ensure that we cover various deployment models as well as the requirements of diverse, real-world applications.

Problem Domain (PD): This problem domain serves as a model of applications that are based on the Resource Description Framework (RDF), which provides a cohesive foundation for agile data integration, for evolving applications as business requirements change, and for delivering to knowledge workers a greater understanding of the information they posses (Feigenbaum 2012). The problem domain uses the parameter values given in Table 4.3 as inputs to Hybridizer. Recall that, Hybridizer uses an I/O based approach for estimating the
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance Metric</td>
<td>Overall Query Execution Time</td>
</tr>
<tr>
<td>Data Disclosure Risk</td>
<td>No. of sensitive nodes exposed to Public cloud</td>
</tr>
<tr>
<td>Resource Allocation Cost</td>
<td>Cloud - Elastic</td>
</tr>
<tr>
<td>Data Model</td>
<td>RDF</td>
</tr>
<tr>
<td>Data Partitioning Model</td>
<td>Vertical</td>
</tr>
<tr>
<td>Data Replication Model</td>
<td>Partial</td>
</tr>
<tr>
<td>Workload Model</td>
<td>SPARQL queries in batch form</td>
</tr>
<tr>
<td>Sensitivity Model</td>
<td>Predicate-based</td>
</tr>
</tbody>
</table>

running time of a task \( t \in W \). An I/O based approach uses selectivity estimation techniques to estimate the input and output sizes of an operator \( \rho \in t \). While selectivity estimation techniques have been widely studied for the relational model, they have not been fully examined in case of RDF. Therefore, in Appendix A we discuss the framework adopted by Hybridizer for performing RDF selectivity estimation.

### 4.7 Performance Evaluation

This section presents results of experiments that were conducted to validate the effectiveness of solutions to \( WDP \). We first present details of our settings followed by the experiments.

#### 4.7.1 Experimental Benchmark

The experiments were conducted using the SP\(^2\)Bench benchmark \cite{Schmidt:2009} for the problem domain defined above. SP\(^2\)Bench has been set in the DBLP \cite{Ley:2006} library scenario, and consists of both, a data generator for creating arbitrarily large DBLP-like documents, and a set of benchmark queries.
4.7.2 Experimental Setup

We conducted experiments on two clusters, the first was located at UT Dallas while the second was located at UC Irvine. The first comprised 14 nodes and was used as the private cloud. The second comprised 28 nodes and was used as the public cloud. A node in the private cloud consists of a Pentium IV processor with \(\approx 290\text{GB}-320\text{GB}\) disk space and 4GB of main memory, while a node in the public cloud consists of an AMD Dual-Core processor with \(\approx 631\text{GB}\) disk space and 8GB of main memory. The clusters were set up using Hadoop v1.0.4, HBase v0.94.2 and Jena-HBase (see Chapter 6).

4.7.3 Experimental Metric Settings

In this subsection, we describe how various metrics associated with \(WDP\) were computed/estimated. In addition, we outline the settings used for the metrics in the experiments.

Statistics collection: The statistics gathering module analyzed a SP\(^2\)Bench dataset containing \(25M\) triples and generated equi-width histograms for every predicate in the SP\(^2\)Bench graph. We used the data types, int, double and string to represent the data. We also created a data type ‘date’ that allows us to represent various dates from the dataset. The number of partitions used in a histogram was supplied as a user-defined value.

Query Workload: The experiments used an SP\(^2\)Bench dataset containing \(25M\) triples and a query workload of 30 queries containing SP\(^2\)Bench queries Q1, Q2, Q3a, Q3b, Q3c, Q9 and Q10. Further, we assumed that each query was equally likely in the workload. The predicates in the queries were randomly modified to vary the range (as mandated by SP\(^2\)Bench) of data that was accessed.

Estimation of weight \(w_x\): The weight \(w_x\) is calculated as the number of I/O operations that can be performed per second on the public and private clouds. We estimated SP\(^2\)Bench-specific weights \(w_{\text{pub}}\) and \(w_{\text{priv}}\) by running queries from SP\(^2\)Bench on our hybrid cloud infrastructure. Then, \(w_{\text{pub}}\) (resp. \(w_{\text{priv}}\)) was computed as the average ratio of the I/O operations
required by the queries on the public side (resp. private side) to the total time required to run all queries on the public side (resp. private side). We executed the SP²Bench queries on a dataset containing 25M triples and estimated \( w_{pub} \) and \( w_{priv} \) to be 1419886.88184779 bytes/sec (\( \approx 13.5\)MB/sec) and 476857.996396746 bytes/sec (\( \approx 4.5\)MB/sec). A larger value for \( w_{pub} \) indicates that our public cloud has a higher I/O throughput than our private cloud.

**Computation of resource allocation cost:** The resource allocation cost was computed using unit prices from Amazon Web Services. We specifically used Amazon S3 pricing to determine storage ($0.140/GB + PUT) and communication ($0.120/GB + GET) costs, where the price for PUT and GET operations is $0.01/1000 requests and $0.01/10000 requests respectively. Also, we used Amazon EC2 and EMR pricing to calculate the processing cost ($0.085 + $0.015 = $0.1/hour). Finally, we estimated the total public cloud resource allocation cost, \( PUB\_MAX\_COST \), by shipping the entire dataset and computation to the public side, as \( \approx 16K \) for SP²Bench, using the previously defined values. We then defined \( PRA\_CONST \) as a fraction (25%, 50%, 75% and 100%) of \( PUB\_MAX\_COST \).

**Definition of sensitive data disclosure risk:** We defined the sensitive data disclosure risk using the following two-part strategy: (1) We defined an overall sensitivity level for the dataset using a predicate-based sensitivity model [Chaudhuri et al. 2007; Miklau and Suciu 2007]. For SP²Bench, we assumed that names of persons and abstracts of all publications are always sensitive while fractions of journals along with their titles and years in which they were published are made sensitive (\( \approx 1\% \) or 5% or 10% of journals are made sensitive). This gives us several different overall sensitivity levels that are then used in the experiments. (2) We defined \( DISC\_CONST \) as being a fraction of the overall sensitivity level for a given dataset, i.e., \( DISC\_CONST \) is varied between 0-100% of the overall sensitivity level defined earlier. For example, in the case when the sensitivity level for journal’s is defined to be 10%, we varied \( DISC\_CONST \) between 0% (none of the 10% sensitive triples should be exposed) and 100% (all of the 10% sensitive triples may be exposed).
Determination of overall workload running time: We determined the overall workload running time, namely $PRI\_RUN\_MAX$, by executing the entire SP$^2$Bench query workload on the private cloud. Using this strategy, the overall workload running time was determined to be $\approx 29000$ seconds. We then defined $DEADLINE\_CONST$ as a fraction (25%, 50%, 75% and 100%) of $PRI\_RUN\_MAX$.

Usage of IBM CPLEX Optimizer: The main aim of the experimental evaluation of Hybridizer was to gauge the performance of the dynamic programming algorithms presented earlier. Towards this end, we compared the performance of the algorithms with IBM’s CPLEX Optimizer which is ideally suited for solving WDP variants since they are formulated as optimization problems. We used a three step procedure for conducting experiments using the CPLEX optimizer: (1) Optimization Model Generation: We converted the three WDP variants into corresponding optimization models using the Optimization Programming Language (OPL) which is part of the IBM ILOG CPLEX Optimization Studio. (2) Data Specification: For each run in an experiment, a data file containing the data to be used by the optimization model corresponding to the experiment was created. (3) CPLEX Optimizer Execution: The CPLEX Optimization Studio was used to run the CPLEX Optimizer on an optimization model and data file corresponding to a particular experiment.

4.7.4 Experiments for Outsourcing WDP variant

In this subsection, we present a performance comparison between the Dynamic Programming (DP) algorithm described earlier and the CPLEX Optimizer for the Outsourcing variant of WDP for the Problem Domain (PD) discussed above.

---

3http://goo.gl/fkz475
4http://goo.gl/gZi1gg
5http://goo.gl/Sf3UQM
Since the aim of this variant is to minimize the disclosure risk, we first computed the risk (in terms of number of triples) when the entire dataset is outsourced to a public cloud (denoted with a suffix “Max” in graphs). The experiment uses this case as a baseline to compare the performance of the dynamic programming algorithm vs. the CPLEX Optimizer.

To perform this experiment, we varied the parameters as follows: (1) The resource allocation cost was varied between 25-100% of the \textit{PRA\_CONST} value that was defined earlier. (2) Two sensitivity levels were defined: 1%-Sensitivity and 10%-Sensitivity (1% and 10% of journals were made sensitive). (3) We defined four different performance deadlines as 25% (the entire workload must be executed within 25% of the time taken for workload execution on a private cloud), 50%, 75% and 100%.

We then computed the disclosure risk for different combinations of these three parameters, the results of which are presented in Figure 4.3. One of the first observations that can
be made is that the results produced with the Dynamic Programming approach are very similar to the ones produced using the CPLEX Optimizer. Also, we observe that as sensitivity level increases, the data exposure level also increases for Dynamic Programming and CPLEX optimizers. Further, one observes that as the performance deadline increases, lesser computations need to be moved to a public cloud, thereby reducing the overall disclosure risk. Additionally, Figure 4.3 shows that a larger monetary budget does not lower disclosure risks, since at least some computations need to be assigned to public clouds to meet performance deadlines, thereby always resulting in some amount of sensitive data disclosure.

4.7.5 Experiments for Cloudbursting WDP variant

In this subsection, we present a performance comparison between the Dynamic Programming (DP) algorithm described earlier and the CPLEX Optimizer for the Cloudbursting variant of WDP for the Problem Domain (PD) discussed above.

Since the aim of this variant is to minimize the resource allocation cost, we first computed the cost (in $) when the entire dataset and workload is outsourced to a public cloud (denoted as “Public” in graphs). The experiment uses this case as a baseline to compare the performance of the dynamic programming algorithm vs. the CPLEX Optimizer. To perform this experiment, we varied the parameters as follows: (1) Four different performance deadlines were defined as 25% (the entire workload must be executed within 25% of the time taken for workload execution on a private cloud), 50%, 75% and 100%. (2) Two sensitivity levels were defined: 1%-Sensitivity and 10%-Sensitivity (1% and 10% of journals were made sensitive). (3) We defined six different sensitive data exposure levels as 0% (none of the sensitive data is exposed), 25%, 40%, 50%, 75% and 100% (all of the sensitive data may be exposed).

We then computed the resource allocation cost for different combinations of these three parameters, the results of which are presented in Figure 4.4. We observe that increasing the performance deadlines results in lower resource allocation costs, since lesser computations
need to be assigned to public clouds to meet performance deadlines, thereby resulting in lower monetary expenditure on public cloud services.

4.7.6 Experiments for Hybrid WDP variant

In this subsection, we present a performance comparison between the Dynamic Programming (DP) algorithm described earlier and the CPLEX Optimizer for the Hybrid variant of WDP for the Problem Domain (PD) discussed above.

Since the aim of this variant is to minimize the overall workload execution time, we first computed the running time of the query workload when all computations are performed on the private cloud (denoted as “Private” in graphs). The experiment uses this case as a baseline to compare the performance of the Dynamic Programming algorithm vs. the CPLEX Optimizer. To perform this experiment, we varied the parameters as follows: (1)
The resource allocation cost was varied between 25-100% of the PRA CONST value that was defined earlier. (2) Two sensitivity levels were defined: No-Sensitivity (the entire dataset is non-sensitive) and 5%-Sensitivity (5% of journals were made sensitive). (3) We defined six different sensitive data exposure levels as 0% (none of the sensitive data is exposed), 25%, 40%, 50%, 75% and 100% (all of the sensitive data may be exposed).

We then computed the overall workload execution time for different combinations of these three parameters, the results of which are presented in Figure 4.5. As before, we observe that the results produced with the Dynamic Programming approach are very similar to the ones produced using the CPLEX Optimizer. Further, one observes that when a user takes additional risks by storing more sensitive data on the public side, one can gain a considerable speed-up in overall execution time (even greater than 50%). Additionally, Figure 4.5 shows that a larger monetary budget further allows one to significantly improve
overall performance (even greater than 50%). This is expected since when more resources are available on the public side, one can better exploit the parallelism that is afforded by a hybrid cloud. Thus, the intuition that a hybrid cloud improves performance due to greater use of inherent parallelism is justified.
CHAPTER 5
REAL-TIME AIS – STORMRIDER: SECURE STORAGE, RETRIEVAL AND ANALYSIS OF SOCIAL NETWORKS

One of the keys to effectively combating terror lies in the ability to securely share information in real-time, especially when an organization or nation is under physical and/or cyber attack. In this context, an AIS implementation that allows secure, real-time information sharing fosters an enhanced situational awareness, especially for intelligence/defense agencies, thereby allowing rapid decisions to be made that could help save thousands of lives. In this chapter, we define the underpinning of such an AIS implementation, namely StormRider, which combines existing Cloud Computing and Semantic Web technologies to provide developers with tools that securely store, query and analyze large-scale, evolving networks in real-time. Using StormRider, one could develop an AIS implementation that allows cooperating parties to securely manage network data in real-time. The rest of the chapter is organized as follows: In Section 5.1 we present the motivation underlying the development of StormRider. This section also presents the novel contributions in StormRider. Section 5.2 presents the architectural details of StormRider. This section also highlights sample Storm topologies that have been implemented for the Twitter network. These topologies demonstrate how one could securely store, query and analyze the Twitter network as the underlying data changes. Finally, a comprehensive performance evaluation of StormRider is given in Section 5.3.

5.1 Motivation

In today’s chaotic times, it is imperative for cooperating agencies, especially governmental agencies, to be able to securely share information in real-time. Towards this end, there
have been several calls by governmental agencies as well as private companies for developing solutions that curtail cyber attacks (Albanesius 2013; Higgins 2012; Lawton 2012). Furthermore, the government already has in place solutions such as the Distributed Common Ground System (DCGS), which provides real-time access to intelligence, surveillance and reconnaissance data for U.S. combat forces deployed throughout the world (Lockheed-Martin 2012). In this context, developing an AIS implementation over existing frameworks such as DCGS would provide secure, real-time information sharing capabilities that produce high quality data, which consequently results in timely decision making. Now, information networks form a major category of data available in the intelligence domain, since they can model a number of domain-specific problems such as communication networks, terrorist cells and information flow networks. To develop an AIS implementation that operates over information networks, one needs to develop an underpinning that allows cooperating parties to securely share, query and analyze large-scale, evolving networks in real-time.

In this chapter, we present StormRider\textsuperscript{1} (Khadilkar et al. 2012)\textsuperscript{2}, a system that uses existing Cloud Computing and Semantic Web technologies to securely store/query evolving networks. StormRider also allows one to perform centrality estimation on an underlying evolving network. Furthermore, StormRider ensures that only authorized users gain access to network data by employing policy-based security mechanisms. To the best of our knowledge, StormRider is the first system to allow secure social network management. Our aim is to create a secure, scalable system that supports diverse use cases from the intelligence domain. Towards this end, we use Storm as one of the building blocks of StormRider. Storm allows StormRider to automatically store, query and analyze evolving networks. In addition, we use a custom-built RDF storage framework, Jena-HBase, to store and query network data

\begin{itemize}
  \item \textsuperscript{1}https://github.com/vaibhavkhadilkar/stormrider
  \item \textsuperscript{2}We acknowledge that some of the preliminary details about StormRider were presented in Khadilkar et al. 2012, which is available at: http://dl.acm.org/citation.cfm?id=2188118.
\end{itemize}
in RDF/SPARQL. We also use the idea of views materialized in HBase to allow faster computation of centrality metrics of nodes. Finally, we employ Discretionary Access Control (DAC) and redaction policies to permit only authorized users to interact with a network.

5.1.1 Our Contributions

StormRider provides the following:

- Allows secure storage, query and analysis of large-scale, evolving networks in real-time using various policy-based access control mechanisms.

- Enables application of features such as inference, reification, property-path queries, etc. to novel use cases defined over evolving networks.

- Performs some fundamental analysis, viz., centrality estimation, on evolving networks.

- Provides application programmers with simple interfaces with which they can store, query and analyze networks of their choice.

5.2 StormRider Architecture

In this section, we begin by presenting an overview of the architecture employed by StormRider. This is followed by a detailed description of the various operations currently supported by StormRider for interacting with social networks.

5.2.1 Architectural Overview of StormRider

StormRider offers a simple abstraction for different ways in which a user can interact with a social network, as its central interface (Model-SN Interface). This abstraction can be used to store, query and analyze different social networks. The main contribution of StormRider is a rich API for manipulating social networks. In particular, the API provides
various tools for interacting with networks, \textit{e.g.}, storage and retrieval of social networks using various Semantic Web representations, basic social network analysis in the form of centrality estimation, \textit{etc}. The key architectural goals of StormRider are as follows:

- The ability for an application programmer to be able to store and analyze networks of their choice by implementing simple interfaces.

- Allow easy access to a few fundamental social network analysis metrics such as centrality computation.

- Open up the Semantic Web space for storage and retrieval of social networks, thereby allowing reasoning algorithms to be applied to them so that they may reveal richer patterns that were previously undetectable.
Figure 5.1 presents an architectural overview of StormRider. User applications (the Application block) can interact with an abstract social network Model (the Model-SN block) that translates high-level user operations (viz. store, query and analyze) on a social network into low-level operations on underlying network representations used by StormRider. The low-level operations are implemented in Storm (Add-Topology, Query-Topology and Analyze-Topology blocks) and are designed to support evolving social networks. However, applications can also use the same topologies to work with single snapshots of a network. An Access Control Layer implemented on top of the topologies provides secure access to data. The Storm framework internally interfaces with the Storage Layer (Jena-HBase), through the Jena-HBase Interface, and the View Layer (HBase tables used as materialized views), through the HBase View Interface, to execute topologies on networks stored in them. We next present a detailed description of the various layers.

5.2.2 Storage Layer

This layer, comprising Jena-HBase, is used to store networks in a RDF representation in a cloud-based framework. The storage of networks in RDF when combined with ontologies defined in other languages (viz. RDFS and OWL) opens up the possibility of using reasoning algorithms on networks to infer previously hidden information from them. The use of a cloud-based framework provides advantages such as scalability, reliability, fault-tolerance, etc. An extensive summary of Jena-HBase is given in Chapter 6.

5.2.3 View Layer

This layer comprises HBase tables that store metadata about nodes in a network. The metadata is used to facilitate a speed-up in performance during the analysis of a network. In StormRider, we currently focus on centrality estimation, particularly degree, closeness and betweenness, as a part of analyzing a social network. Since closeness/betweenness centrality
are path-based metrics, we use a variant of the landmark technique \cite{Potamias2009,Rattigan2006} for fast shortest path estimation as a part of the analysis topology implemented in Storm. We present a brief overview of the landmark technique and our adaptation of it, followed by a detailed description of the HBase views.

The landmark technique for shortest path estimation involves selecting a set of landmarks and computing the shortest distance from all other nodes to these nodes offline \cite{Potamias2009}. The idea is to represent each non-landmark as a vector of shortest path distances to the set of landmarks. Then at runtime, when the shortest distance between a pair of nodes is needed, it is estimated using the precomputed vectors. Additionally, in reference \cite{Potamias2009}, the authors prove that the landmark selection problem is NP-Hard and they subsequently present different landmark-selection strategies such as Random, Degree and Centrality, along with constrained and partitioned variants of them.

In StormRider, we made the following change to the idea of maintaining a vector of shortest path distances to landmarks: For every non-landmark, we store its nearest landmark along with the shortest distance to that landmark rather than a vector of distances to all landmarks. This simplification allows us to better manage shortest paths in an evolving network, while maintaining relatively accurate results for shortest path estimation between any two nodes. Additionally, we use the Degree strategy as our landmark-selection strategy since it strikes the right balance between approximation quality and computational efficiency.

Figures 5.2 and 5.3 present the Node-centric and Landmark-centric views that are constructed for each network link that a user wants to analyze. Note that, in the figures below the values in the first column (NodeId and LandmarkId&NodeId) are used as row keys in the actual implementation of the views as HBase tables.

Figure 5.2 presents an example of a Node-centric view. This view stores metadata for every node of the network as a distinct tuple in the underlying HBase table. Each node in the network is represented by assigning its unique identifier to a distinct row key (viz. NodeId) in the table. This metadata includes the following:
Figure 5.2. HBase View: Node-centric information

- **Adjacency List**: This column family stores the adjacency list for every node.

- **Metric information**: This column family stores information about the various metrics (viz. degree, closeness and betweenness centrality) currently supported by StormRider for every node. Each metric (viz. DegC, CloseC and BetC) is represented as a separate column under the “metric” column family and stores the current value of that metric.

- **Landmark information**: This column family stores landmark-related information for a given node. This includes whether a given node is a landmark (Is-Landmark column), the distance from a given node to its nearest landmark (Dist-To-Closest-Landmark column) and a given node’s closest landmark (Closest-Landmark column).

Figure 5.3 presents an example of a Landmark-centric view. This view stores information that is specific to each node that has been currently designated as a landmark. The row key in this view is to be interpreted as follows: The first part (viz. LandmarkId) denotes a designated landmark, while the second part (viz. NodeId) denotes a non-landmark that is connected to the given landmark through some path. Additionally, every tuple in this view represents path-related information on how a non-landmark is connected to a landmark. This path-related information includes the following:

<table>
<thead>
<tr>
<th>Node Id</th>
<th>Adj-List</th>
<th>Metric</th>
<th>Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Is-Landmark</td>
</tr>
<tr>
<td>sub1</td>
<td>sub2 sub3</td>
<td>0.75</td>
<td>0.5</td>
</tr>
<tr>
<td>sub10</td>
<td>sub20 sub30</td>
<td>0.1</td>
<td>0.03</td>
</tr>
</tbody>
</table>
Figure 5.3. HBase View: Landmark-centric information

- Node-specific information: This column family stores path-related information for paths from a non-landmark to a designated landmark. This includes information about the number of hops between a non-landmark and its closest landmark (Distance column), the number of distinct paths between the two nodes (Num-Of-Paths column) and the actual paths, enumerated as a sequence of nodes, between them (Paths column).

5.2.4 Access Control Layer

This layer uses a two-fold measure to allow authorized users to access data. The first measure uses two types of access control policies to permit authorized users to manage networks, which are enforced in the spouts of topologies.

The first type of policy uses the Discretionary Access Control (DAC) model (Samarati and di Vimercati, 2000) to permit authorized users to interact with a network. DAC policies can be described using the access control matrix model from (Harrison et al., 1976), which captures currently held authorizations in a system as a matrix. The state of the system is defined by a triple \((S, O, A)\), where \(S\) denotes subjects who can exercise privileges; \(O\) denotes objects, on which privileges can be exercised; and \(A\) is the access matrix, where rows correspond to \(S\), columns correspond to \(O\), and \(A[s, o]\) reports the privileges of \(s \in S\) on \(o \in O\) (Samarati and di Vimercati, 2000). The kinds of objects and their associated actions depend on a particular system. Although the matrix model succinctly captures authorization
concepts, it cannot be implemented as is, due to its large size and sparse nature. Instead, the matrix is stored in column form as an *Access Control List* (ACL), such that each object *o* has an associated list, indicating for each subject, the actions that can be performed on *o*.

```
ex:resource1 vocab:hasPolicy ex:policy1 ;
ex:policy1 vocab:hasName "Policy 1" ;
  vocab:hasType "Access Control" ;
  vocab:hasAuth "WRITE" ; vocab:hasUser "John" ;
```

StormRider uses an access control mechanism comprising two steps: (1) A user submits an ACL encoded as an RDF graph such as the one shown above for the Add-Topology, where *ex:resource1* represents a named graph that will contain Twitter data, while *ex:policy1* represents a policy that allows John to load data into *ex:resource1*. (2) At runtime, the Add-Topology spout checks if the current user is authorized to load data into *ex:resource1* by checking the user’s credentials against the submitted ACL. A similar policy definition/enforcement mechanism is used by spouts of the Query-/Analyze-Topologies to allow only authorized users to execute queries or conduct analyses on a network.

The second type of policy uses *pattern matching* to restrict the sub-network that can be accessed by a user, and is therefore only applicable to Query-Topologies. Furthermore, the policy type is divided into the following three categories, based on the kinds of restrictions that they enforce.

1. **Class-specific restrictions**: This policy type allows users to access only instances of a specific class. To illustrate, as a result of enforcing the policy shown below, John is only allowed to access the sub-network comprising instances of type *ex:Person* in his queries.

```
ex:resource1 vocab:hasPolicy ex:policy2 ;
ex:policy2 vocab:hasName "Policy 2" ;
  vocab:hasAuth "READ" ; vocab:hasUser "John" ;
  vocab:hasMatchPattern "?s rdf:type ex:Person" ;
```
(2) **Property-specific restrictions**: This policy type allows users to access the sub-network belonging to a specific property. To illustrate, as a result of enforcing the policy shown below, John is only allowed to access the sub-network belonging to the property `tw:Has_Friend` in his queries.

```rdfs
ex:resource1 vocab:hasPolicy ex:policy3 ;
ex:policy3 vocab:hasName "Policy 3" ;
   vocab:hasAuth "READ" ; vocab:hasUser "John" ;
   vocab:hasMatchPattern "?s tw:Has_Friend ?o" ;
```

(3) **Path-based restrictions**: This policy type allows users to only access specific paths in a network. To illustrate, as a result of enforcing the policy shown below, John is only allowed to access Jim's neighborhood covering his friends and friends-of-friends. Such a restriction is particularly important in the context of government agencies, where analysts can now only search networks based on pre-discovered surveillance (*viz.* the possibility of Jim being a threat), rather than conducting open-ended searches (*viz.* considering all users as threats), thereby ensuring the privacy of non-threatening individuals.

```rdfs
ex:resource1 vocab:hasPolicy ex:policy3 ;
ex:policy3 vocab:hasName "Policy 3" ;
   vocab:hasAuth "READ" ; vocab:hasUser "John" ;
   vocab:hasMatchPattern "<ex:Jim> tw:Has_Friend/tw:Has_Friend ?o" ;
```

The concept of restricting access based on classes, properties and paths provides users with a fine-grained and flexible strategy for enforcing access control. In practice, StormRider enforces restrictions using the following strategy: (1) A user submits access restrictions encoded as RDF graphs such as the ones shown above. (2) At runtime, a Query-Topology enforces the restrictions conditionally, based on user credentials, the search patterns specified in a SPARQL query, and applicable policies.
Note that, one could also create access restrictions based on the characteristics of the sub-network being accessed by a user. For example, one could create an access restriction that does not allow a user to access a sub-network, produced as a result of executing a SPARQL query, if the number of triples in the sub-network exceeds a threshold.

The second measure protects sensitive portions of a query result before it is shared with a user using redaction policies, which are enforced in a special bolt in the Query-Topology.

A redaction policy $p$ sanitizes sensitive parts of results (viz. sanitize nodes – $\text{SNode}$, edges – $\text{SEdge}$, and paths – $\text{SPath}$) before releasing them to a user. The process of redaction (also called sanitization) can be formalized as follows: Given a sanitization operation $o \in O$, where $O = \{\text{SNode}, \text{SEdge}, \text{SPath}\}$, in a policy $p$ and an RDF graph $g$, we define a sanitization function $S : O \times G \rightarrow G$ that takes graph $g \in G$ and operation $o \in O$ and produces a sanitized graph $g' \in G$, such that $g'$ satisfies $o$, written as $o(g')$.

```sparql
ex:resource2 vocab:hasPolicy ex:policy1 ;
ex:policy1 vocab:hasName "Policy 1" ;
   vocab:hasType "Redaction" ;
   vocab:hasOp "SNode ( ?s foaf:name ?o )" ;
```

In StormRider, redaction is performed using the REDACT framework [Rachapalli et al., 2013], and comprises the following steps: (1) A user submits redaction policies such as the one shown above, where $\text{ex:resource2}$ denotes the graph on which the policy is defined, while $\text{SNode ( ?s foaf:name ?o )}$ denotes an operation that sanitizes all names in $\text{ex:resource2}$. (2) At runtime, the special Query-Topology bolt checks if operations in any policy are applicable to current results. An applicable policy is identified based on user credentials and the submitted query. If an applicable policy is identified, the bolt applies the operation in the policy to the results.

In the future, we plan to broaden the scope of this layer to include additional types of data security mechanisms.
5.2.5 Topology Layer

The Topology layer comprises various topologies through which a user interacts with an evolving network. A user needs to implement application-specific topologies for interacting with networks. Then, StormRider’s API executes these topologies on their behalf. Thus, StormRider can execute topologies that interact with any network. However, to demonstrate the efficacy of StormRider, we have used the Twitter network as the basis for the construction of sample Add-, Query- and Analyze-Topologies, each of which is described next.

Add-Topology for Twitter Dataset

An Add-Topology allows an authorized user to store a network of their choice in Jena-HBase. Since StormRider only provides a sample Add-Topology for the Twitter network, application programmers need to define their own Add-Topologies for storing social networks of their choice. StormRider’s API then executes these topologies on their behalf.

Figure 5.4 shows the design of a sample Add-Topology that adds Twitter data to StormRider. The general idea is to have a spout that allows authorized application users to read data from Twitter using the policy-based mechanism described earlier and then uses the data for multiple purposes. If the user is authorized, various pieces of information such as screen name, location and friends are collected for a randomly selected user, and then emitted as tuples to Storage and View bolts. The data is converted into an RDF representation and stored in Jena-HBase (denoted by the Storage block). At the same time, the data is also used to update the views (denoted by the Node-centric View and Landmark-Information blocks) maintained in the view layer. Note that, in this work we only analyze the “friendship” link between Twitter users. Therefore, both views are created/maintained only for this link. We have divided the discussion about the Add-Topology into two parts, namely the spout that reads Twitter data, and the bolts that store Twitter data as RDF and update the Node-/Landmark-centric views. Additionally, to have a clear understanding
of how our system works, we have presented the discussions on the spout as well as the bolts as a series of algorithms, followed by detailed explanations. Before we begin this discussion, we give some notes that apply to all algorithms presented throughout this chapter.

Algorithmic Notes: All pseudocode given in the algorithms below is executed as a particular method, viz., `nextTuple()` in a spout or `execute(Tuple input)` in a bolt. Additionally, for algorithms that describe bolts, any input parameter other than a `tuple`, is always constructed/initialized by the `prepare()` method of a bolt. Therefore, these additional parameters are simply shown as inputs to algorithms for simplicity and convenience.

Add-Topology - Spout

As stated earlier, a Storm spout, **Twitter-API-Spout**, allows an authorized application user to read data from Twitter, which is then injected into various bolts that update the storage and view layers. We begin by presenting a description of the Twitter-API-Spout followed by explanations for the different bolts that update the storage and view layers.
Algorithm 4 Twitter-API-Spout()

Input: $U, M_s, Q, M$

1: $authorized \leftarrow \text{verify}(U, M_s)$
2: if $authorized$ then
3: Pick a random Twitter user
4: Get information specific to this user such as name, location, tweets, friends, followers etc. using the Twitter API
5: for each piece of information do
6: Emit a tuple containing that information, in the form (user-identifier, property, value), to the Jena-HBase-Bolt
7: end for
8: Emit a tuple with the user-identifier to the Update-Nodes-View-Bolt
9: $usersAndNeighbors \leftarrow \text{execute}(Q, M)$
10: for $userAndNeighbor \in usersAndNeighbors$ do
11: Emit a tuple containing the user-identifier in $userAndNeighbor$
12: end for
13: end if

Algorithm 4 presents a set of steps used by Twitter-API-Spout to allow authorized users to read data from Twitter, which is injected into various bolts used to update the storage and view layers. The algorithm begins by verifying whether the current user is authorized to read data from Twitter (line 1). If the user is authorized, the algorithm randomly selects a Twitter user (line 3). Algorithm 4 then retrieves information such as name, friends, etc. for the selected user. (line 4), and emits a tuple for each piece of information to Jena-HBase-Bolt (lines 5-7). After that, the algorithm emits a tuple containing only the current user's identifier to Update-Nodes-View-Bolt (line 8). Finally, for every friend of the current user stored in the system, the algorithm repeatedly emits a tuple containing the friend's identifier to Count-User-Degree-Bolt (lines 9-12). In practice, this task is performed by executing the following query over the underlying Jena-HBase model,

```
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>

SELECT ?x ?y
WHERE { ?x twitter:Has_Friend ?y }
```

and emits a new tuple for each unique binding of values to variables $?x$ and $?y$. 
Add-Topology - Bolts

We now present a detailed discussion of various bolts implemented as a part of the Add-Topology that are used to update the storage and view layers.

The Storage Layer Bolt: This layer consists of a single bolt, Jena-HBase-Bolt, that takes tuples containing user-specific information emitted by Twitter-API-Spout, converts them into RDF triples, and stores the triples in an underlying Jena-HBase model.

Algorithm 5 Jena-HBase-Bolt()
Input: tuple, M
1: subject ← tuple.getString(0)
2: predicate ← tuple.getString(1)
3: object ← tuple.getString(2)
4: M.add(triple(subject, predicate, object))

Algorithm 5 presents a set of steps used by Jena-HBase-Bolt to add Twitter data to an underlying Jena-HBase model. For every tuple received, the algorithm extracts a string representation of the three constituent parts of this tuple, which correspond to the subject, predicate and object respectively of the triple that can be generated for the tuple (lines 1-3). The algorithm first constructs this corresponding triple using these components, and then adds the triple to the underlying Jena-HBase model (line 4).

The View Layer Bolts: This layer consists of multiple bolts that update the Node-centric and the Landmark-centric view with metadata pertaining to nodes of the Twitter network. The Update-Nodes-View-Bolt takes the identifier of the user currently selected by Twitter-API-Spout as an input, and updates the Node-centric View with metadata related to this user. Note that, this bolt only updates information for non-landmarks, while the Landmark-Information block updates information associated with landmarks. The sequence of steps used by this bolt to perform this task is presented below in Algorithm 6.
Algorithm 6 \textsc{Update-Nodes-View-Bolt()}

\textbf{Input:} \textit{tuple, Q, M, HBase-Nodes-View}

1: \textit{node} \leftarrow \textit{tuple.getString}(0)
2: \textit{isLandmarkNode} \leftarrow \textit{lookup(node, HBase-Nodes-View)}
3: \textbf{if} \textit{isLandmarkNode} \textbf{== false} \textbf{then}
4: \hspace{1em} \textit{adjList} \leftarrow \textit{execute(Q, M)}
5: \hspace{1em} \textit{HBase-Nodes-View.add(node, Adj-List, adjList)}
6: \hspace{1em} \textit{closest-landmark} \leftarrow \textit{BFS(node)}
7: \hspace{1em} \textit{HBase-Nodes-View.add(node, Is-Landmark, N)}
8: \hspace{1em} \textit{HBase-Nodes-View.add(node, Dist-To-Closest-Landmark, closest-landmark.distance)}
9: \hspace{1em} \textit{HBase-Nodes-View.add(node, Closest-Landmark, closest-landmark.node)}
10: \textbf{end if}

The algorithm begins by extracting a string representation of the node currently under consideration by Twitter-API-Spout into variable \textit{node} (line 1). The algorithm next looks-up the Node-centric view and determines if the current \textit{node} is a landmark (line 2). If it is not, then the algorithm updates the Node-centric view with metadata related with the current non-landmark. The algorithm first queries the underlying model using the following query,

\texttt{PREFIX ex: <http://www.example.org/twitter#>}

\texttt{PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>}

\texttt{SELECT ?y}

\texttt{WHERE \{ ex:node twitter:Has_Friend ?y \}}

that retrieves all Twitter users connected to the current user, \texttt{<ex:node>}, by a “friendship” link (line 4). This list of users, denoted by variable \textit{adjList}, is then added to the “Adj-List” column family in Node-centric view for the current user (line 5). The algorithm then determines the closest landmark by running the breadth-first-search (BFS) algorithm from the current \textit{node} (line 6). The result of the BFS algorithm is stored in a data structure, “closest-landmark”. This data structure stores both, the distance to the closest landmark (in closest-landmark.distance) and the user-identifier of the closest landmark (in closest-landmark.node). Next, the algorithm adds/updates the following column families in Node-centric view with this information (lines 7-9): (1) “Is-Landmark” is updated with the value
“N”, since the current node is not a landmark. (2) “Dist-To-Closest-Landmark” is updated with the value in closest-landmark\( .distance \). (3) “Closest-Landmark” is updated with the user-identifier in closest-landmark\( .node \).

The **Landmark-Information** block in Figure 5.4 updates information associated with landmarks. This block consists of multiple components, which are described below.

The **Landmark-Selection** block is used to select the top-\( k \) nodes from amongst all nodes using the Degree strategy given in [Potamias et al., 2009], and are subsequently stored as a list of landmarks. Note that, multiple executions of the Landmark-Selection block could lead to changes to this list. The first bolt in this block, **Count-User-Degree-Bolt**, takes as input a tuple comprising a user-identifier and computes a count of all friends (i.e. degree for the “friendship” link) for that user. Algorithm 7 presents the sequence of steps required to perform this operation. This bolt uses a field grouping on user-identifier to ensure that tuples for a given user, emitted by Twitter-API-Spout, always go to the same bolt task.

**Algorithm 7** **Count-User-Degree-Bolt()**

**Input:** tuple, hashMap \(< user Id, Long >\)

**Output:** opTuple

1: \(node \leftarrow \) tuple.getString(0)
2: \(deg \leftarrow \) hashMap.get(node)
3: \(\text{if } deg == \text{null } \text{then}\)
4: \(\text{hashMap.put(node, 1)}\)
5: \(\text{else}\)
6: \(\text{hashMap.put(node, deg + 1)}\)
7: \(\text{end if}\)
8: \(\text{opTuple.add(node, hashMap.get(node))}\)
9: \(\text{Emit opTuple}\)

Algorithm 7 begins by extracting a string representation of the user-id from the incoming tuple (line 1). The algorithm then looks-up this id in the hash-map, and tries to retrieve the current user’s degree (line 2). If the degree is null, the user does not exist in the hash-map and is therefore added into it with a degree of 1 (line 4). On the other hand, if the user
does exist in the hash-map, that user’s degree is incremented by 1 (line 6). Finally, a tuple comprising the current node and it’s degree is forwarded to Rank-Users-Bolt (lines 8-9).

The remaining bolts in the Landmark-Selection block are part of a design pattern in Storm that allows us to select the top-$k$ landmarks based on degrees of all nodes in the network. The broad idea is to divide the task of selecting the top-$k$ landmarks between two bolts as follows: (1) The first bolt finds top-$k$ nodes within separate partitions of the stream. (2) The second bolt then merges the different top-$k$ lists into a global list of top-$k$ landmarks. This approach ensures a scalable solution even in the presence of very large streams.

The Rank-Users-Bolt is used to find the top-$k$ landmarks within separate partitions of the stream. The sequence of steps used to perform this task is outlined below in Algorithm 8. This bolt makes use of a field grouping on the user-identifier of a node so that tuples for the same node are always sent to the same bolt task.

Algorithm 8 Rank-Users-Bolt()

Input: tuple, $k$, list

Output: opTuple

1: node ← tuple.getString(0)
2: nodePos ← list.find(node)
3: if nodePos == null then
4:   list.add(tuple)
5: else
6:   list.add(nodePos, tuple)
7: end if

8: Sort list in decreasing order based on the degrees of nodes
9: if list.size > $k$ then
10:   Remove tuple at position $k$
11: end if
12: opTuple.add(list)
13: Emit opTuple

Algorithm 8 receives as input a tuple consisting of a user-id for a node and its current count of “friends”. Additionally, the algorithm also receives the value $k$ which denotes an upper bound for number of landmarks, and a list that holds the top-$k$ landmarks. Algo-
Algorithm begins by extracting the user-id from the tuple (line 1). Next, the algorithm tries to find the position, \( \text{nodePos} \), of node in \( \text{list} \) (line 2). If the node is not part of \( \text{list} \), the tuple containing the node is added to the end of \( \text{list} \), while if the node is present, the tuple is added to \( \text{list} \) at position \( \text{nodePos} \) (lines 3-7). The algorithm then sorts tuples in \( \text{list} \) in decreasing order based on the degrees (viz. number of friends) associated with nodes (line 8). If the size of \( \text{list} \) exceeds upper bound \( k \), the tuple at position \( k \) is removed since at any given time the maximum size of \( \text{list} \) can be \( k + 1 \) (lines 9-11). Finally, the algorithm constructs a new tuple consisting of the updated \( \text{list} \) and emits this tuple to Merge-Users-Bolt (lines 12-13).

The \textbf{Merge-Users-Bolt} merges the various lists of top-\( k \) landmarks generated by Rank-Users-Bolt into a global list of top-\( k \) landmarks. The pseudocode used to perform this task is given below in Algorithm [9]. This bolt makes use of a global grouping on lists generated by Rank-Users-Bolt to ensure that all top-\( k \) lists are sent to a single bolt task.

**Algorithm 9 Merge-Users-Bolt()**

**Input:** tuple, \( k \), list

**Output:** \( \text{opTuple} \)

1. \( \text{nodeList} \leftarrow \text{tuple.getString(0)} \)
2. for \( l \in \text{nodeList} \) do
3. \( \text{nodePos} \leftarrow \text{list.find(l)} \)
4. if \( \text{nodePos} == \text{null} \) then
5. \( \text{list.add(l)} \)
6. else
7. \( \text{list.add(nodePos, l)} \)
8. end if
9. Sort \( \text{list} \) in decreasing order based on the degrees of nodes
10. if \( \text{list.size} > k \) then
11. Remove elements in \( \text{list} \) for \( k \leq i < \text{list.size()}) \)
12. end if
13. end for
14. \( \text{opTuple.add(list)} \)
15. Emit \( \text{opTuple} \)

Algorithm [9] receives as input a tuple comprising some top-\( k \) list generated by one of the tasks of Rank-Users-Bolt. Additionally, the algorithm also receives, the value \( k \) which
denotes an upper bound for the number of landmarks to be used in the global list, and a list that holds the global top-\(k\) landmarks. Algorithm 9 begins by extracting the list, nodeList, from the tuple (line 1). Then, for every node \(l\) in nodeList, the algorithm repeats a sequence of steps (lines 3-12), similar to the steps used in Algorithm 8 to update list, the global list of top-\(k\) landmarks. The only difference being that the size of list could be greater than \(k + 1\). Therefore, the algorithm removes all nodes between positions \(k\) and \(list.size() - 1\) (line 11). Finally, the algorithm constructs a new tuple consisting of the list of global top-\(k\) landmarks and emits this tuple to Update-Landmarks-Information-Bolt (lines 14-15).

The Update-Landmarks-Information-Bolt is used to update the Landmark-centric view with metadata regarding shortest path information on how non-landmarks are connected with landmarks. Additionally, this bolt also updates Node-centric view with metadata pertaining to each landmark in the top-\(k\) list. Algorithm 10 presents a list of successive steps that can be used to perform these tasks.

**Algorithm 10 UPDATE-LANDMARKS-INFORMATION-BOLT()**

**Input:** tuple, \(Q, M, nonLandmarksList\), HBase-Nodes-View, HBase-Landmarks-View

1: \(list \leftarrow\) tuple.get(0)
2: for lmark \(\in\) list do
3: Clear rows in HBase-Landmarks-View that contain lmark
4: landmarkInfo \(\leftarrow\) SSSP(lmark)
5: for node \(\in\) nonLandmarksList do
6: \(nInfo \leftarrow\) landmarkInfo.find(node)
7: HBase-Landmarks-View.add(lmark\&node, Distance, nInfo.distance)
8: HBase-Landmarks-View.add(lmark\&node, Num-Of-Paths, nInfo.numPaths)
9: for \(i = 1\) to numPaths do
10: HBase-Landmarks-View.add(lmark\&node, Paths, nInfo.getPath\((i)\))
11: end for
12: end for
13: adjList \(\leftarrow\) execute\((Q, M)\)
14: HBase-Nodes-View.add(node, Adj-List, adjList)
15: HBase-Nodes-View.add(lmark, Is-Landmark, Y)
16: HBase-Nodes-View.add(lmark, Dist-To-Closest-Landmark, 0)
17: HBase-Nodes-View.add(lmark, Closest-Landmark, lmark)
18: end for
Algorithm 10 receives as input a tuple comprising a list of global top-k landmarks. The algorithm also receives as input the query $Q$ that was used earlier to retrieve the adjacency list of a node. Further, Algorithm 10 receives a Jena-HBase model $M$ that stores network data. Finally, the algorithm also receives the list of non-landmarks, $nonLandmarksList$, and Node-centric and Landmark-centric views, namely HBase-Nodes-View and HBase-Landmarks-View. Note that, $nonLandmarksList$ can be generated by querying Node-centric view for all nodes that contain a value “N” for the “Is-Landmark” column family. The algorithm begins by extracting the global list of top-k landmarks from the input tuple (line 1). For every landmark in this list, $lmark$, the algorithm first clears rows in HBase-Landmarks-View that contain $lmark$ (line 3). Then, the algorithm executes the single-source-shortest-path (SSSP) algorithm for $lmark$ (line 4). This algorithm computes shortest paths from $lmark$ to all other nodes in the network and stores the results in a data structure, $landmarkInfo$. The algorithm then iterates over $nonLandmarksList$ and in each iteration, the algorithm first finds path-related information for the current node ($node$), which is stored in a node-specific data structure, $nInfo$ (line 6). Algorithm 10 then updates HBase-Landmarks-View with path-related information for $node$ (lines 7-11). This information includes the following: (1) The distance between $lmark$ and $node$ is stored in column family “Distance”. (2) The number of paths between $lmark$ and $node$ are stored in column family “Num-Of-Paths”. (3) The actual paths between $lmark$ and $node$ are stored as sequences of nodes in column family “Paths”. Finally, the algorithm adds/updates the following column families in Node-centric view with information pertaining to $lmark$ (lines 13-17): (1) “Adj-List” is updated with the adjacency list of $lmark$, which is obtained by executing the previously described query $Q$ on $M$. (2) “Is-Landmark” is updated with value “Y” since the current node is a landmark node. (3) “Dist-To-Closest-Landmark” is updated with value “0”. (4) “Closest-Landmark” is updated with value $lmark$, since the node under consideration is itself a landmark.

As stated earlier, we have only presented a sample Add-Topology for Twitter. Application programmers wishing to use StormRider need to implement their own topologies for
storing networks of their choice. However, they can reuse spouts and bolts implemented in StormRider within their custom Add-Topologies. We now move on to a detailed discussion of the Query-Topology that has been implemented for Twitter.

**Query-Topology for Twitter Dataset**

A Query-Topology allows authorized users to execute queries (continuous or ad-hoc) on an underlying network as data from the network is being streamed into Jena-HBase. StormRider’s existing Query-Topology can be used to execute SPARQL queries on any underlying network. Additionally, application developers can implement their own topologies for querying networks stored in underlying storage and view layers. In this way, StormRider allows various types of query topologies to be executed over underlying network data. We now present a few sample topologies that can be currently handled by StormRider.

**SPARQL Query-Topology**

A SPARQL Query-Topology allows authorized users to execute a SPARQL query (continuous or ad-hoc) over RDF data stored in Jena-HBase. A mechanism similar to that described for the Add-Topology ensures that only authorized users are allowed to execute queries. A Query-Topology that executes a query, called SPARQL-Query-Topology, consists of a spout called **SPARQL-Query-Spout** that takes the input query and executes it over the network stored in Jena-HBase. Next, the results are passed on to **Redaction-Policy-Applier-Bolt**, which is a special bolt used to apply redaction policies to results using the mechanism described earlier. After the bolt performs redaction on query results, the sanitized results are passed on to **HBase-Result-Table-Loader-Bolt**, which loads them into an HBase table, after which they can be streamed to a client. Figure 5.5 depicts the execution strategy of SPARQL-Query-Topology.
We now present some sample SPARQL queries and the pseudocode used by SPARQL-Query-Spout to execute them. These queries demonstrate that StormRider can not only support general-purpose queries but also special-purpose use cases such as automatically querying evolving networks and retaining and accessing prior states of a network.

**Use Case 1:** Consider the following query that aims to find all users who have listed their current location as “Richardson, TX”. This type of query can be classified as ad-hoc, since it will most likely be executed only when one wants to find out about other users that live in Richardson, TX, which will usually be an infrequent request.

PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>

SELECT ?x
WHERE { ?x twitter:Location "Richardson, TX" } 

**Use Case 2:** Consider a more complex use case that aims at tracking the evolution of a user’s neighborhood over time. Such requests are common in police surveillance and targeted advertising. The use of Semantic Web technologies in StormRider easily allows us to accomplish this task by using SPARQL path queries. This type of query can be classified as continuous, since a user would want to track another user’s neighborhood over an extended period of time. Furthermore, such a query requires tracking various paths that emerge from the node under consideration and as such can be effectively captured by a regular-expression
SPARQL query. Towards this end, we use the Gleen library \cite{Detwiler:2008} to define such regular-expression SPARQL queries. This library is closely integrated with the Jena framework and extends SPARQL with support for defining regular path queries over Jena models. Since Jena-HBase simply acts as an HBase-backed storage subsystem for Jena, we can directly use Gleen to define regular path queries over models stored in Jena-HBase. The following query sample can be used to track the neighborhood of a user “John” covering his friends as well as friends-of-his-friends (2 hops). The SPARQL-Query-Topology will automatically execute this query periodically so that the neighborhood of the user “John” is tracked over an extended period of time. Note that, 2 hops is simply used for illustrative purposes, and in practice, StormRider can support queries with an arbitrary number of hops.

PREFIX ex: <http://www.example.org/twitter#>
PREFIX tw: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x
WHERE { <ex:John> gleen:onPath("/[tw:Has_Friend]/[tw:Has_Friend])" ?x) }

Use Case 3: We now present a use case that allows users to store and access all versions of an evolving network. This request is relevant for intelligence agencies who want to trace the evolution of a potential terrorist cell. The idea of storing and accessing prior snapshots is similar to ImmortalDB, which builds transaction time support into SQL Server Engine \cite{Lomet:2005}. This feature is automatically enabled in StormRider through the use of an RDF representation and the SPARQL query language. However, the way a user interacts (viz. stores and accesses) with a network needs to be slightly modified as given below.

**Timestamping and Version management:** Since StormRider stores network data as RDF, a triple $t_i$ can be reified with a timestamp $T_i$ that denotes the construction time of $t_i$. Any subsequent changes to $t_i$ (e.g. re-discovery or updates) lead to insertion of a new triple, $t_j$, reified with timestamp $T_j$ indicating the construction time of $t_j$. Now, $T_j > T_i$, since $t_j$
was discovered later than $t_i$. In this way, we store two versions of the triple, $t_i$ and $t_j$, which can be accessed with the appropriate timestamp ($T_i$ or $T_j$). Similarly, when a triple is to be deleted, a special version of the triple is inserted, reified with a timestamp that denotes when it was deleted. The task of version control does not require any additional special functions. Consider the following example where we discover that “John” has a friend “James”. Note that, the following prefixes are used in subsequent RDF triples,

ex: <http://www.example.org/twitter#>

twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>

This relationship is captured in the following triple,

ex:John twitter:Has_Friend ex:James

which is stored in a Jena-HBase model. Additionally, the triple is reified with the timestamp of when it was discovered. This leads to addition of the following triples to the same model,

ex:John twitter:Has_Friend ex:James

ex:triple123 rdf:type rdf:Statement
ex:triple123 rdf:subject ex:John
ex:triple123 rdf:predicate twitter:Has_Friend
ex:triple123 rdf:object ex:James
ex:triple123 twitter:Timestamp "2012-01-25T21:00:00"^^xsd:dateTime

Suppose that we later discover that “John” has two friends, namely “James” (as before) and “Joe”. The re-discovery of John’s relationship with James and the discovery of a new relationship between John and Joe leads to addition of the following triples to the model,

ex:John twitter:Has_Friend ex:James
ex:triple123 rdf:type rdf:Statement
ex:triple123 rdf:subject ex:John
ex:triple123 rdf:predicate twitter:Has_Friend
ex:triple123 rdf:object ex:James
ex:triple123 twitter:Timestamp "2012-01-25T21:00:00"^^xsd:dateTime

ex:triple124 rdf:type rdf:Statement
ex:triple124 rdf:subject ex:John
ex:triple124 rdf:predicate twitter:Has_Friend
ex:triple124 rdf:object ex:James
ex:triple124 twitter:Timestamp "2012-01-26T22:00:00"^^xsd:dateTime

ex:John twitter:Has_Friend ex:Joe
ex:triple125 rdf:type rdf:Statement
ex:triple125 rdf:subject ex:John
ex:triple125 rdf:predicate twitter:Has_Friend
ex:triple125 rdf:object ex:Joe
ex:triple125 twitter:Timestamp "2012-01-26T22:00:00"^^xsd:dateTime

To enable this feature of historical storage of network data, a user only needs to reify triples being added to Jena-HBase in their custom-built Add-Topologies.

**SPARQL syntax**: The SPARQL query language itself does not need to be extended in any way to provide access to historical versions of a network in StormRider. A user can query older snapshots of the network by making use of the FILTER construct of SPARQL with appropriate timestamp values. This is in sharp contrast to ImmortalDB in which changes were made to the data definition language to include an Immortal attribute that allows the definition of transaction-time tables. Additionally, ImmortalDB supported historical queries through the use of an “AS OF” clause with the “Begin Transaction” statement. For example,
the following SPARQL query could be used to find users who were connected with the user “John” through the “friendship” link at any point in time prior to and including a timestamp value of "2012-01-25T21:00:00"^^xsd:dateTime,

PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>

SELECT ?z
WHERE { ?x twitter:Timestamp ?y .
    ?x rdf:subject <ex:John> .
    ?x rdf:predicate <twitter:Has_Friend> .
    ?x rdf:object ?z .
    FILTER( ?y <= "2012-01-25T21:00:00"^^xsd:dateTime ) }

Thus, to enable historical queries of network data, a user only has to define the appropriate SPARQL query that makes use of a FILTER construct along with an appropriate timestamp value that captures the time period they want to query.

**Use Case 4**: Consider a use case that allows users to continually search and monitor publicly available social media information. This use case is relevant to intelligence agencies who wish to collect and analyze publicly available social media information to enhance their situational awareness. Towards this end, several agencies (*viz.* FBI, IARPA, DARPA, etc.) have sought the development of applications that monitor social media websites ([FBI](#) [2012]; [IARPA](#) [2011]; [DARPA](#) [2011]). StormRider can provide automated support for this use case, since it contains a topology for executing continuous queries. Additionally, we can implement a custom-built Add-Topology that monitors information as triples are being added to Jena-HBase. Furthermore, StormRider can support the following variations of the basic use case:

**4a**: Instantly search and monitor specific keywords and strings posted to online social media websites. For example, the following SPARQL query can be used to continuously search for Twitter users that post tweets containing the keyword “bombing”.
PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE { ?x twitter:Has_Tweet ?y .
   FILTER regex(?z, "bombing", "i") }

4b: Conduct word combination searches that monitor numerous keywords at the same time. For example, the following query continuously searches for users that post tweets containing the keyword “bombing” or “explosion” followed by “America”.

PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE { ?x twitter:Has_Tweet ?y .
   FILTER regex(?z, ".*(bombing|explosion).*america.*", "i") }

4c: Conduct word combination searches that monitor numerous keywords for a particular location. The following query continuously searches for users that post tweets with a location “Richardson,TX” and contain the keyword “bombing” or “explosion” followed by “America”.

PREFIX ex: <http://www.example.org/twitter#>
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?y
WHERE { ?x twitter:Has_Tweet ?y .
   ?y twitter:Place_Full_Name "Richardson,TX"^^xsd:string .
   FILTER regex(?z, ".*(bombing|explosion).*america.*", "i") }
We now present a set of algorithms that can be used to execute any query (continuous or ad-hoc) on a network stored in Jena-HBase. Algorithm 11 presents pseudocode implemented in SPARQL-Query-Spout, which executes a query on an underlying network. Algorithm 12 shows pseudocode implemented in Redaction-Policy-Applier-Bolt, which enforces access control by applying redaction policies to results. Finally, Algorithm 13 describes pseudocode used in HBase-Result-Table-Loader-Bolt, which loads the redacted query results into a user-specified HBase table, after which these results can be streamed to a client.

Algorithm 11 SPARQL-QUERY-SPOUT()
\[
\text{Input: } U, M_S Q, M, \text{interval, prevTime, numOfReports, maxReports}
\]
\[
\text{Output: } r
\]
1: \( \text{authorized} \leftarrow \text{verify}(U, M_S) \)
2: \text{if} \ authorized \text{ then}
3: \quad \text{if} \ \text{numOfReports} < \text{maxReports} \text{ then}
4: \quad \quad \text{currTime} \leftarrow \text{getCurrentTime}()
5: \quad \quad \text{if} \ (\text{currTime} - \text{prevTime}) \geq \text{interval} \text{ then}
6: \quad \quad \quad \text{execute}(Q, M)
7: \quad \quad \quad \text{for} \ \text{each result} \ r \in \text{R} \ \text{do}
8: \quad \quad \quad \quad \text{Emit a tuple containing} \ r \ \text{to HBase-Result-Table-Loader-Bolt}
9: \quad \quad \text{end for}
10: \quad \quad \text{numOfReports} \leftarrow \text{numOfReports} + 1
11: \quad \quad \text{prevTime} \leftarrow \text{currTime}
12: \quad \text{end if}
13: \text{end if}
14: \text{end if}

Algorithm 11 takes as input, user credentials \( U \), a special-purpose model \( M_S \), a query \( Q \), a Jena-HBase model \( M \), an interval (in seconds) that specifies the time period after which \( Q \) needs to be executed, the time at which the algorithm last executed \( Q \), \( \text{prevTime} \), the number of times \( Q \) has been executed, \( \text{numOfReports} \), and the maximum number of times, \( \text{maxReports} \), the user wants to execute \( Q \). The algorithm begins by verifying whether the current user is authorized to execute a query (line 1). If the user is authorized, the algorithm checks if \( \text{numOfReports} \) is less than \( \text{maxReports} \) (line 3). If it is, the algorithm updates
Algorithm 12 Redaction-Policy-Applier-Bolt()

**Input:** $U$, $Q$, $M_S$, tuple

**Output:** $opTuple$

1. $opTuple \leftarrow $ tuple
2. policies $\leftarrow$ getPolicies($U$, $Q$, $M_S$)
3. for policy $\in$ policies do
4. \hspace{1em} $opTuple \leftarrow$ applyPolicy(policy, $opTuple$)
5. end for
6. Emit $opTuple$

Algorithm 12 takes as input, user credentials $U$, a query $Q$, a special-purpose model $M_S$ and a tuple containing a single row of result $R$ computed by Algorithm 11. The algorithm begins by assigning the result tuple to $opTuple$ (line 1). Next, the algorithm retrieves applicable policies based on credentials $U$ and query $Q$ from $M_S$ (line 2). The algorithm then iterates over the applicable policies and applies them sequentially to $opTuple$ (lines 3-5). Once all policies are applied, the algorithm emits the redacted tuple, $opTuple$ (line 6).

Algorithm 13 HBase-Result-Table-Loader-Bolt()

**Input:** tuple, HBase-Result-Table

1. numOfVars $\leftarrow$ tuple.size()
2. for $i \leftarrow$ 0 to numOfVars $- 1$ do
3. \hspace{1em} colVal $\leftarrow$ tuple.getString($i$)
4. \hspace{1em} HBase-Result-Table.add($i$, resultVar, colVal)
5. end for

Algorithm 13 receives as input, a tuple containing a single row of the redacted result $R$ that was computed by Algorithm 12. Additionally, the algorithm also receives the user-
specifies a table, HBase-Result-Table, which will store $R$. The algorithm begins by computing the number of variables, $numOVar$, in $Q$, since the input tuple will contain $numOVar$ values in every result $r$ (line 1). The algorithm then iterates $numOVar$ times to add values obtained from the input tuple to the corresponding $resultVar$ column families in HBase-Result-Table (lines 2-5). Then, a client application can access $R$ by iterating over rows of HBase-Result-Table.

**Custom-built Query-Topology**

As stated earlier, StormRider’s architecture allows users to define custom-built Query-Topologies for query tasks specific to their own networks. In this section, we present an example of one such custom-built Query-Topology that is constructed for the Twitter network. Figure 5.6 presents an overview of this topology whose goal is to periodically output the top-$k$ users whose tweets are most often retweeted. Note that, the design of this topology is similar to the one that computes the top-$k$ landmarks from amongst all nodes that we presented earlier. Therefore, in this section we simply give a description of the topology while omitting the algorithms, since they are similar to the ones we presented earlier.

Figure 5.6 presents a sample topology used to execute the task outlined above. As shown in the figure, the topology consists of multiple components. A spout, User-Retweet-Spout,
periodically generates tuples consisting of a Twitter user-id and the number of retweets for every tweet of that user. To perform this task, the spout uses the following SPARQL query:

```sparql
PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>
SELECT ?x ?z
WHERE { ?x twitter:Has_Tweet ?y .
  ?y twitter:Tweet_Retweet_Count ?z . }
```

Every unique binding of values to variables ?x and ?z is used to produce a tuple that is sent to Count-User-Retweet-Bolt. The **Count-User-Retweet-Bolt** count all retweets for a user-id that was output by User-Retweet-Spout. This bolt uses a field grouping on user-id to ensure that tuples with the same user-id always go to the same task. Finally, this bolt emits the user-id and the rolling count of retweets for that user as its output to Rank-Users-Bolt.

The **Rank-Users-Bolt** performs parallel computations to find many top-k users across partitions of the stream. This bolt uses the same user-id based field grouping that was used earlier. The different top-k lists generated over the entire cluster are merged into a global top-k list using **Merge-Users-Bolt**. This bolt uses a global grouping to ensure that all top-k lists are sent to a single task that can correctly merge these lists into the final top-k users list, which is forwarded to the client application. The rationale behind this approach is to find top-k users within separate partitions of the stream, followed by a global merging of these individual top-k’s to obtain the global top-k users across the entire stream. This two-phase process ensures a scalable solution even in the presence of very large streams.

**Analyze-Topology for Twitter Dataset**

An Analyze-Topology allows authorized users to analyze networks stored in Jena-HBase. StormRider provides a sample Analyze-Topology that allows authorized users to perform centrality estimation on evolving networks. Developers can also implement custom Analyze-Topologies to analyze networks stored in the underlying storage layer. They can also use the
Figure 5.7. An Analyze-Topology used to compute centrality metrics on the Twitter network metadata stored in the view layer to speed-up their analyses if they have used the custom bolts defined earlier as a part of their Add-Topologies.

Figure 5.7 shows the design of a sample Analyze-Topology that computes centralities of nodes from the Twitter network. The idea is to use a spout that allows authorized users to periodically generate a list of all nodes currently stored in the system, and then emits each node as a tuple to different bolts that perform centrality estimation (viz. degree, closeness and betweenness) for them. As before, we have divided the discussion into its constituent parts, namely a spout that generates tuples containing user-id’s of nodes, and bolts that perform centrality estimation. Additionally, we present algorithms for each component for a better understanding. Finally, an access control mechanism similar to the one described earlier is used to ensure that only an authorized user is allowed to generate a list of users.

**Analyze-Topology - Spout**

A spout, **Twitter-User-Spout**, is used to periodically allow authorized users to generate a list of all Twitter users currently stored in the system. Then, the spout constructs a tuple for each user in the list, which is then forwarded to different bolts in the topology. We now present an algorithm that implements the task that we just outlined.
Algorithm 14 Twitter-User-Spout()

Input: $U$, $M_S$, $Q$, $M$, interval, $prevTime$

1: $authorized \leftarrow \text{verify}(U, M_S)$
2: if $authorized$ then
3: \hspace{0.5em} $currTime \leftarrow \text{getCurrentTime}()$
4: \hspace{1em} if $(currTime - prevTime) \geq \text{interval}$ then
5: \hspace{1.5em} $R \leftarrow \text{execute}(Q, M)$
6: \hspace{2em} for $r \in R$ do
7: \hspace{2.5em} Emit a tuple containing the user-identifier in $r$ to all bolts
8: \hspace{1em} end for
9: \hspace{0.5em} $prevTime \leftarrow currTime$
10: end if
11: end if

Algorithm 14 takes as input, user credentials $U$, a special-purpose model $M_S$, a query $Q$, a Jena-HBase model $M$, an interval (in seconds) specifying the time period after which $Q$ is to be executed, and the last time when a list of users was generated, $prevTime$. The algorithm first verifies if the current user is authorized to generate a list of users (line 1). If they are authorized, the algorithm updates $currTime$ with the current time (line 3). The algorithm then checks if an interval amount of time has passed since the list was last generated (line 4). If it has, the algorithm re-executes $Q$ on model $M$. In practice, $Q$ is defined as follows:

PREFIX twitter: <http://cs.utdallas.edu/semanticweb/StormRider/twitter#>

SELECT DISTINCT ?x

WHERE { ?x twitter:Has_Friend ?y }

This query retrieves all Twitter users and stores them in variable $R$. Then, for every node $r \in R$, the algorithm emits a tuple containing the user-id of that node to all bolts in the topology (line 7). Additionally, the algorithm updates $prevTime$ to $currTime$ (line 9).

Analyze-Topology - Bolts

We now present a description of bolts implemented in the Analyze-Topology, that perform centrality estimation for currently stored Twitter users.
The **Degree-C-Bolt** is used to compute degree centrality for all nodes currently stored in the system. Algorithm 15 presents a sequence of steps used to perform this task.

**Algorithm 15 Degree-C-Bolt()**

**Input:** tuple, totalNodes, HBase-Nodes-View

1. node ← tuple(getString(0))
2. degree ← 0
3. adjList ← HBase-Nodes-View.get(node, Adj-List)
4. for adjNode ∈ adjList do
5. degree ← degree + 1
6. end for
7. degC ← \frac{degree}{totalNodes−1}
8. HBase-Nodes-View.add(node, DegC, degC)

Algorithm 15 takes as input a tuple containing a user-id, the total number of users, `totalNodes`, and the Node-centric view, HBase-Nodes-View. The algorithm begins by extracting a string representation of the user-id, `node`, from the input tuple (line 1). In addition, the algorithm also initializes a variable, `degree` to 0 (line 2); `degree` denotes the number of “friendship” links user has with other users. Then, the algorithm obtains the adjacency list, `adjList`, for `node` by using HBase-Nodes-View (line 3). For every node, `adjNode`, in this list the algorithm increments `degree` by 1 (line 5). Algorithm 15 then computes degree centrality of the current node, `degC`, as a fraction of `degree` over `totalNodes − 1` (line 7). The term `totalNodes − 1` is used since the “friendship” link is undirected. Finally, the algorithm updates the Node-centric view with the `degC` value (line 8).

The **Closeness-C-Bolt** computes the closeness centrality for all currently stored Twitter users. Algorithm 16 presents a series of steps that are used to perform this task.

Algorithm 16 receives the following as input: (1) A tuple containing a user-id. (2) The total number of users, `totalNodes`. (3) The Node-centric view, HBase-Nodes-View. (4) The Landmark-centric view, HBase-Landmarks-View. The algorithm first extracts the user-identifier of the current user into `node` (line 1) and also initializes `sumPaths` to 0 (line 2); `sumPaths` denotes the sum of paths from `node` to all other nodes. The algorithm then
Algorithm 16 Closeness-C-Bolt()

Input: tuple, totalNodes, HBase-Nodes-View, HBase-Landmarks-View

1: node ← tuple.getString(0)
2: sumPaths ← 0
3: closestLandmark ← HBase-Nodes-View.getNode(node, Closest-Landmark)
4: if node == closestLandmark then
5: end if
6: else
7: initDist ← HBase-Nodes-View.getNode(node, Dist-To-Closest-Landmark)
8: end if
9: landmarkAndNodes ← HBase-Landmarks-View.getNode(closestLandmark)
10: for lmarkAndNode ∈ landmarkAndNodes do
11: sumPaths ← sumPaths + HBase-Landmarks-View.getNode(lmarkAndNode, Distance) + initDist
12: end for
13: closeC ← \frac{totalNodes - 1}{sumPaths}
14: HBase-Nodes-View.add(node, CloseC, closeC)

finds the landmark closest to node (closestLandmark) by performing a look-up in HBase-Nodes-View (line 3). Next, the algorithm initializes initDist to 0 if node is the same as closestLandmark, else it is initialized with the distance between node and closestLandmark, which can be obtained from HBase-Nodes-View (lines 4-8). The algorithm then retrieves all rows containing closestLandmark from HBase-Landmarks-View (line 9). For each row in this set, the algorithm increments sumPaths with the initDist from node to closestLandmark and the distance from closestLandmark to other non-landmarks (lines 10-12). Recall that, a row key in HBase-Landmarks-View comprises a LandmarkId and a NodeId, where the NodeId represents some non-landmark. Then, Algorithm 16 computes the closeness centrality of the current node, closeC, as a fraction of totalNodes – 1 over sumPaths (line 13). Finally, the algorithm updates Node-centric view with the closeC value (line 14).

The Betweenness-C-Bolt computes betweenness centrality for currently stored Twitter users. Algorithm 17 presents a series of steps used to perform this task.

Algorithm 17 receives the same inputs as Algorithm 16. The algorithm first extracts the identifier of the current user into nodeToCompute (line 1) and also initializes betweenness
Algorithm 17 BETWEENNESS-C-BOLT()

Input: tuple, totalNodes, HBase-Nodes-View, HBase-Landmarks-View

1: nodeToCompute ← tuple.getString(0)
2: betweenness ← 0
3: allNodes ← HBase-Nodes-View.getRowKeys()
4: for node ∈ allNodes do
5:   closestLandmark ← HBase-Nodes-View.get(node, Closest-Landmark)
6:   landmarkAndNodes ← HBase-Landmarks-View.get(closestLandmark)
7:   for landmarkAndNode ∈ landmarkAndNodes do
8:     numPaths ← HBase-Landmarks-View.get(landmarkAndNode, Num-Of-Paths)
9:     listOfPaths ← HBase-Landmarks-View.get(landmarkAndNode, Paths)
10:    numOfPathsWithNode ← 0
11:   for path ∈ listOfPaths do
12:     if path.contains(nodeToCompute) then
13:       numOfPathsWithNode ← numOfPathsWithNode + 1
14:     end if
15:   end for
16:   betweenness ← betweenness + \(\frac{\text{numOfPathsWithNode}}{\text{numPaths}}\)
17: end for
18: betC ← \(\frac{\text{betweenness} \times 2}{(n-1) \times (n-2)}\)
19: HBase-Nodes-View.add(node, BetC, betC)

to 0 (line 2); betweenness denotes the betweenness for the current user. Next, the algorithm retrieves all rows from HBase-Nodes-View (line 3). For each node in this set, the algorithm retrieves the landmark closest to node (closestLandmark) by performing a look-up in HBase-Nodes-View (line 5). This is followed by retrieving all rows from HBase-Landmarks-View that contain closestLandmark (line 6). For every row in this set, the algorithm retrieves the number of paths (numPaths) to the non-landmark (line 8), represented by the NodeId part of the row key of HBase-Landmarks-View. Also, all actual paths (listOfPaths) are retrieved (line 9). For every path (path) in this list, the algorithm checks if path contains nodeToCompute. If it does, the algorithm increments numOfPathsWithNode, which tracks the number of paths containing nodeToCompute (line 13). Once the algorithm iterates over all available paths to a non-landmark, the algorithm updates betweenness with the fraction of numPaths that contain nodeToCompute, namely numOfPathsWithNode (line 16). Fi-
nally, after the algorithm iterates over all nodes, it normalizes the value of \textit{betweenness} to compute betweenness centrality, $betC$ (line 19), which is stored in HBase-Nodes-View (line 20). Again, the normalization factor is $\frac{(n-1) \times (n-2)}{2}$, since the network is undirected.

5.3 Performance Evaluation

This section presents details of our experimental investigation into the performance of StormRider. We begin by presenting details of the experimental setup. We also give a brief description of the dataset used in our experiments. We then present our findings that detail the effectiveness of the various topologies implemented in StormRider.

5.3.1 Experimental Setup

We conducted our experiments on a cluster containing 14 nodes. Each machine in the cluster consists of a Pentium IV processor with 290GB to 360GB disk space and 4GB main memory. The cluster ran Hadoop v0.20.2 and HBase v0.90.1. The nodes are connected with a 48-port Cisco switch on an internally created private network.

The experiments compared the landmark technique for centrality estimation used in StormRider with the exact method from [Brandes, 2001]. The experiments for exact centrality computation were conducted on a machine that used an Intel Xeon processor with a 2TB hard drive and 32GB memory.

5.3.2 Experimental Dataset - Twitter

To demonstrate the efficacy of StormRider, we used Twitter as the basis of the implementation for various Add-, Query- and Analyze-Topologies. Therefore, our experimental investigation into StormRider uses a Twitter dataset, comprising information (profile and tweet) on 10M randomly selected Twitter users. The information for each user was gathered twice, where the interval between information collection was a few months (between 3-6),
thus constructing two snapshots of the network. The selected interval of time ensured clear changes in the network, particularly in the neighborhoods of active Twitter users.

A Twitter profile consists of basic information such as name, location, bio, count of followers/friends, the latest tweet etc. A user’s tweet contains information such as when the tweet was created, the message of the tweet, it’s source if any, etc. For the experiments, we used the Add-Topology to add information to StormRider. Simultaneously, the Analyze-Topology was executed to compute centrality metrics for each user.

5.3.3 Experimental Procedures

In this subsection, we present details of the procedures used in the two experiments we carried out with StormRider.

The first experiment compared the performance of the landmark-based approximation method for estimating shortest paths between nodes with the exact method from [Brandes, 2001]. The results of shortest path computations were in turn used to compute closeness and betweenness centrality for a maximum of 10M Twitter users. We did not evaluate degree centrality since one can always compute an exact value using a node’s adjacency list and the total nodes. Also, we did not evaluate other metrics, since we wanted to build a prototype that supports automated addition, retrieval and basic analysis of a network. In the future, we plan to evaluate additional metrics. In the experiments, the number of landmarks was randomly set as, total nodes in graph / 100. The Degree strategy from [Potamias et al., 2009] was used to select landmarks, since as shown in Figure 3 of [Potamias et al., 2009], at large orders of magnitude, the Degree technique produces a small approximation error (approx 0.03-0.09). Finally, recall that we focus on the “friendship” link between Twitter users, therefore, our experiments are conducted on an undirected graph.

The second experiment measured the overhead of enforcing redaction policies on query results. We did not measure the overhead of enforcing access control policies since it involves performing a credential verification step or an intersection between search patterns
in a user query and match patterns in an access restriction policy, and thereby requires an inconsequential amount of time.

5.3.4 Performance of StormRider for Closeness Centrality

Aim: The aim of this experiment was to evaluate the efficiency of the HBase views in computing closeness centrality. The evaluation is in turn dependent on the number of landmarks selected and the mechanisms employed to update the views by the Add-Topology.

Procedure: In this experiment, we computed closeness centrality for increasing graph sizes (from 2M to 10M nodes) by using shortest paths obtained through the landmark technique in our implementation, and the exact technique from [Brandes, 2001]. Since we constructed two snapshots of Twitter, we automatically computed closeness centrality for each snapshot in turn using the topologies described earlier. Then, we analyzed the performance of both methods across the following metrics:

1. Approximation Error: The approximation error measures the accuracy of the landmark technique in computing closeness centrality, and is computed as follows: \(|\hat{l} - l|/l\) where \(l\) is the actual closeness centrality and \(\hat{l}\) is the approximation.

2. Overall Execution Time: This metric measures the time required to perform the approximate and exact computation of closeness centrality. The overall execution time for the approximate case is computed as a sum of both, the time to update the views when nodes are re-discovered (Node-centric and Landmark-centric View blocks in Figure 5.4) and the time to perform actual closeness centrality computation (Closeness-C-Bolt).

3. User-centric Metric Time: This metric measures the time required to compute closeness centrality for a single user with the landmark technique across both snapshots. The metric time is estimated as the time required by Closeness-C-Bolt to compute closeness centrality for the selected user. The user was a randomly selected non-landmark node, for whom the number of friends changed from 175 to 202 across both snapshots.
Figure 5.8. Performance of StormRider for Closeness Centrality computation

Note that, the graphs for Approximation Error and Overall Execution Time given below present results of closeness centrality computation over the second Twitter snapshot.

Observations: We now present our observations for the metrics outlined above.

1. Approximate Error: The leftmost graph of Figure 5.8 presents results of the approximation error experiment for StormRider. We see that the error remains small even when the number of users increases from 2M to 10M, as is expected, since even at large orders of magnitude, the error remains small as shown in (Potamias et al., 2009).

2. Overall Execution Time: The graph in the center of Figure 5.8 presents results of the experiment that measures the overall execution time of the approximate algorithm vs. the exact algorithm from (Brandes, 2001). We observe that the execution time of the approximate algorithm is far better than the exact algorithm. This is expected, since the exact algorithm requires $k$ single-source-shortest-path (SSSP) computations while the approximate algorithm requires $k/100$ SSSP computations and $k$ BFS executions, thus leading to a lesser overall execution time (where $k$ is the total number of nodes).

3. User-Centric Metric Time: The rightmost graph of Figure 5.8 presents results of the experiment that measures the time to compute closeness centrality for the preselected user across both Twitter snapshots. Firstly, we observe that the difference between
execution times across both snapshots is very small for the entire range (2M-10M). This is because the two views used by the Closeness-C-Bolt during centrality estimation for the selected user are nearly identical across both snapshots. We also observe that the execution time consistently increases across the entire range (2M-10M). This slight increase is due to a longer lookup time in the Landmark-centric view, which increases in size as the number of users is increased from 2M to 10M.

5.3.5 Performance of StormRider for Betweenness Centrality

*Aim:* The aim of this experiment was to evaluate the efficiency of the HBase views in computing betweenness centrality. Again, the evaluation depends on the number of landmarks and the implementation of different bolts in the Add-Topology for updating the views.

*Procedure:* The procedure was the same as the previous experiment that computes closeness centrality, the only difference being that we compute betweenness centrality in this case.

*Observations:* We now present our observations for the experimental metrics that were outlined above. We also notice that results in Figure 5.9 are very similar to results in Figure 5.8. This is because both, closeness and betweenness centrality use shortest paths in their computation. Furthermore, the algorithms that compute shortest paths remain the same in both of sets experiments, thereby leading to nearly identical results.
1. Approximate Error: The leftmost graph of Figure 5.9 presents results of the approximation error experiment. Again, we see that the error remains small as the number of users increases from 2M to 10M, as is expected, since at large orders of magnitude, the approximation error is small as shown in (Potamias et al., 2009).

2. Overall Execution Time: The graph in the center of Figure 5.9 presents results of the experiment that measures the execution time of the approximate algorithm vs. the exact algorithm from (Brandes, 2001) for betweenness centrality computation. We observe that the execution time of the approximate algorithm is far better than the exact algorithm. This is expected, since the exact algorithm requires $k$ all-pairs-all-shortest-path (APASP) computations while the approximate algorithm requires $k/100$ SSSP computations, $k$ BFS executions and $k^2$ iterations over all nodes in the network, thus leading to a lesser overall execution time (where $k$ is the total number of nodes). Additionally, the execution times are slightly greater than their corresponding counterparts in Figure 5.8. This is because the SSSP computations are now replaced by APASP calculations or multiple iterations over nodes in the network.

3. User-Centric Metric Time: The rightmost graph of Figure 5.9 presents results of the experiment that measures the time to compute betweenness centrality for the preselected user across both Twitter snapshots. Again, we observe that the difference between execution times is small for the entire range (2M-10M). This is because the views used by the Betweenness-C-Bolt are nearly identical. As before, we also see that the execution time consistently increases across the entire range. This is due to a longer lookup time in the Landmark-centric view, which increases in size as the number of users increases.

5.3.6 Overhead of Enforcing Redaction Policies

Aim: The goal of this experiment was to measure the overhead of applying sanitization operations defined in redaction policies to results of user-submitted SPARQL queries.
Procedure: We conducted this experiment on a maximum of 100K users by comparing the running times of SPARQL queries with and without enforcement of sanitization operations. The SPARQL queries used, the associated sanitization operations, and their purpose are presented in Table 5.1. The last column of Table 5.1 shows the number of triples that are sanitized as a part of the given operation for a dataset comprising 100K users.

Figure 5.10 presents results of this experiment when the number of users increases from 25K to 100K. Firstly, we observe that the time to execute the original query increases as the number of users increases. Additionally, the process of applying sanitization operations results in a small overhead, the magnitude of which increases as the number of users increases. Furthermore, the overhead is slightly larger for the SEdge operation as compared with SNode and SPath, since SEdge affects the largest number of triples.
Table 5.1. Queries and Redaction Operations used in Experimental Evaluation; ST denotes Sanitized Triples

<table>
<thead>
<tr>
<th>Original Query</th>
<th>Sanitization Operation</th>
<th>Purpose</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTRUCT { ?s tw:Twitter_Id ?o1. ?s tw:Screen_Name ?o2 } WHERE { ?s tw:Twitter_Id ?o1. ?s tw:Screen_Name ?o2 }</td>
<td>SNode (&quot;s tw:Twitter_Id ?o)</td>
<td>Protect User Id's</td>
<td>100000</td>
</tr>
</tbody>
</table>
CHAPTER 6
REAL-TIME AIS – JENA-HBASE: A DISTRIBUTED, SCALABLE AND EFFICIENT RDF TRIPLE STORE

As stated in the previous chapter, a prerequisite in the development of StormRider is the presence of a scalable RDF storage framework. Such a framework allows one to store/query network data in real-time using RDF/SPARQL, thereby allowing one to support novel use cases such as those described previously. In this chapter, we present Jena-HBase, a framework that uses existing Cloud Computing technologies to construct a distributed, scalable and efficient RDF storage framework. The rest of the chapter is organized as follows: Section 6.1 presents the motivation behind the development of Jena-HBase, and the novel contributions in Jena-HBase. In Section 6.2 we present details of Jena-HBase. In particular, we cover storage layouts that have been implemented and query processing strategies used by each layout. A performance evaluation of Jena-HBase is given in Section 6.3.

6.1 Motivation

RDF data model is a graph data model that uses the notion of a triple, which denotes that a resource (subject) holds a value (object) for some attribute (predicate) of the resource. RDF triples are usually stored in triple stores backed by relational databases or other disk-based mechanisms, that were originally designed for relational or other data models. The simplest way to store RDF triples comprises a relation/table consisting of three columns, one each for subjects, predicates and objects. However, this approach suffers from lack of scalability, since the table is usually located on a single machine, and abridged query performance, since a query requires several self-joins with the same table. There have been
several approaches (Wilkinson et al., 2003a; Broekstra et al., 2002; Harris and Gibbins, 2003; Franz-Inc., 2005) to modify the single-table schema to improve query performance. However, these approaches still suffer from the scalability problem. Therefore, to solve the scalability issue it is only natural to move from a single-machine to multi-machine configuration.

The roadmap taken by triple stores to achieve this transition is similar to that taken by traditional DBMS’s, which evolved from single machine to modern distributed versions that offer advantages such as scalability and enhanced query performance. Cloud computing paradigm has allowed triple stores to mirror what was done by distributed databases for single node databases. Tools such as Hadoop and HBase have made it possible to harness the processing power of multiple machines in parallel. In this chapter, we present Jena-HBase\(^1\) (Khadilkar et al., 2012)\(^2\) a HBase backed triple store that can be used with Jena.

The main aim of Jena-HBase is to mitigate the scalability issue that exists with single machine RDF storage systems, without resulting in a considerable loss of performance. The motivation to opt for Jena is its widespread acceptance, and built-in support for diverse RDF processing features such as: (1) An RDF API that allows manipulation of RDF data in various formats (XML, N-triples, etc.). (2) An Ontology API that supports ontology application development. (3) A query engine that supports the SPARQL query language. Consequently, we only need to implement operations that allow addition, deletion and retrieval of triples from HBase. Additionally, any Jena user could easily adapt their current implementations to use Jena-HBase with minimal code changes. Further, HBase was selected for the storage layer for two reasons: (1) HBase is a column-oriented store and in general, column-oriented stores perform better than row-oriented stores (Abadi et al., 2007). (2) Hadoop comprises Hadoop Distributed File System (HDFS), a distributed file system that stores data, and

\(^1\)https://github.com/vaibhavkhadilkar/hbase-rdf

\(^2\)We acknowledge that some of the preliminary details about Jena-HBase were presented in (Khadilkar et al., 2012), which is available at: http://ceur-ws.org/Vol-914/
MapReduce, a framework for processing large volumes of data stored in HDFS. HBase uses
HDFS as its storage mechanism but does not require MapReduce for accessing data. Thus,
Jena-HBase does not require the implementation of a MapReduce-based query engine, which
is a challenge by itself, for executing queries on RDF triples stored in HBase.

6.1.1 Our Contributions

Jena-HBase framework provides the following:

• A variety of custom-built RDF data storage layouts for HBase that provide tradeoffs
  in terms of query performance/storage.

• Support for features such as reification, inference and SPARQL processing through the
  implementation of appropriate Jena interfaces.

6.2 Jena-HBase Architecture

In this section, we begin with a brief overview of the Jena SDB architecture, which is used
as the foundation of Jena-HBase. This is followed by an architectural overview of Jena-
HBase. Next, we present the storage schema for each layout that is currently implemented
in Jena-HBase. We also discuss the advantages and disadvantages for each of these storage
schemas. Each schema is part of a separate layout and a user can choose a layout by using
a configuration file. Finally, we present the query processing strategy for each layout.

6.2.1 Overview of Jena SDB architecture

Jena SDB is a part of the Jena framework, and is designed to specifically support SPARQL.
A relational database is used to provide storage and SPARQL queries are translated into
SQL queries over the underlying database.

Figure 6.1 presents an architectural overview of Jena SDB, which uses a store to provide data loading and querying capabilities on a relational database. A store represents a single RDF dataset and is contained in a single database. The same store can contain several RDF graphs, each with its own separate layout. All operations on a RDF graph are implicitly converted into operations on the underlying layout. These operations include the following:

- Formatting a layout (i.e. delete all triples while preserving the tables in a layout), which is represented by the Formatter block.

- Loading-unloading triples into a layout (Loader block).

- Querying a layout for triples that match an \(<S, P, O>\) pattern. Since Jena SDB operates on relational databases, the querying mechanism involves translating a SPARQL query into an SQL query, which is subsequently executed. This is accomplished using the Query Compiler, SQL Bridge and SQL Generator blocks from Figure 6.1.
• Additional operations include the following: (1) Maintaining a database connection (Connection block). (2) Using an appropriate database type (Database block). (3) Maintaining configuration information for each RDF graph (Config block).

Jena SDB supports three layouts: (1) A “simple” layout (layout1) that contains a table with four columns, one each for subjects, predicates, objects and graphs. (2) A “hash” layout that contains a table, “Triples”, for the default graph containing three columns, namely subject, predicate and object. A separate table, “Quads”, is created for named graphs that contains a graph column in addition to the subject, predicate and object columns. Instead of storing actual nodes in columns, an integer representing an 8-byte hash of a node is stored with the mapping from integer to value being stored in a separate “Nodes” table. Additionally, a separate table, “Prefixes”, stores a mapping between prefixes for an RDF graph and their actual URI’s. (3) An “index” layout that is exactly the same as the hash layout. The only difference is that integers stored in a column are 4-byte sequence id’s into the Nodes table.

6.2.2 Jena-HBase Storage Architecture

Figure 6.2 shows our adaptation of Jena SDB for storing RDF graphs in HBase. We use a store to abstract users from underlying operations on HBase tables. Also, we do not need all operations provided in Jena SDB. The Connection, Formatter, Loader and Config blocks have the exact same functions as in Jena SDB. We replace the complex query processing components from Jena SDB with a simpler Query Runner that matches an \( <S, P, O> \) pattern with appropriate underlying HBase table(s). We also do not need the Database block from Jena SDB. Since we use HBase tables, we adopt certain naming conventions for them, thereby allowing us to support abstractions for stores and graph types (default vs. named).

Table Naming Conventions: The name of an HBase table comprises three parts:

1. “name” is the name of the store that the current table belongs to.
2. “pre” is the name of the RDF graph that the current table belongs to. A value of “tbl” denotes tables belonging to the default graph in a RDF dataset while named graphs in the dataset use the graph URI as the value.

3. The last part of a table name uses keywords specific to a layout.

Jena-HBase currently support six layouts each of which is presented below. These layouts use a varying number of HBase tables with different table schemas.

**Simple Layout**

Figure 6.2 presents the storage schema for the Simple layout. For every RDF graph, this schema comprises three tables, each respectively indexed by *subjects*, *predicates* and *objects*. Each table consists of a single “triples” column family. For each node used to index a table, a new column is created under “triples” for every triple that contains the indexing node.
The name of a column is made up of nodes other than the indexing node. For example, given a subject $s_1$ in a subjects table, a new column will be created for every triple that contains $s_1$ in a given RDF graph. The name of a column will contain a predicate-object pair, $p_i o_i$, where $p_i$ and $o_i$ are a predicate and object that are present in some triple containing $s_1$. The predicates and objects tables are updated similarly. This schema is not efficient in terms of storage space, since the subject, predicate and object of a triple are stored three times. Moreover, multiple occurrences of the same resource URI or literal are stored more than once across all tables and within a table. However, a find operation does not require lookups into other mapping tables since all information is self-contained within a single table.

**Vertically Partitioned Layout**

Figure 6.4 presents the storage schema for the Vertically Partitioned layout. For every unique predicate in a given RDF graph, this schema creates two tables, respectively indexed by subjects and objects. In addition to the store name and prefix, a table name now includes a predicate name. Each of the two tables contains a single “nodes” column family. For each node used to index a table, a new column is created under “nodes” for every triple containing the indexing node and the predicate that is part of the table name. The name of
Figure 6.4. Vertically Partitioned Layout – Storage Schema

A column is made up of the node other than the indexing node and *predicate* in a triple. For example, given a *subject* $s_1$ in a subjects table, a new column will be created for each triple containing $s_1$ and some predicate $p_1$, which is part of the table name. The name of a column will contain an *object*, $o_i$, where $o_i$ is an *object* belonging to some triple containing $s_1$ and $p_1$. The objects table is updated symmetrically to the subjects table. This schema leads to significant storage space savings when compared to the Simple layout. This is because moving a *predicate* name to the table name completely eliminates the storage of a *predicate* from the subjects and objects tables. Moreover, it completely removes the predicates table that was used in the Simple layout. Nevertheless, multiple occurrences of the same resource URI or literal are stored many times across all tables and within a table. Also, a find operation may now need to lookup multiple tables to ensure that a complete result is returned to a user.

**Indexed Layout**

Figure 6.5 presents the storage schema for the Indexed layout. In this layout, six tables are created for every RDF graph. These tables represent the six possible combinations of nodes ($S$ – subjects, $P$ – predicates, $O$ – objects) in a triple, namely SPO, SOP, OSP, OPS, PSO and POS. The combinations also form the suffixes for table names. Each table contains a
single “blank” column family. In practice, no columns are added to this column family, since a given node ordering (SPO, SOP etc.) is stored in a row key. For example, given triple $s_1 p_1 o_1$, a row is added to each of the six tables. A new row with key $s_1 p_1 o_1$ will be added to the SPO table. Additionally, no new columns will be created under “blank”. The remaining tables (SOP, PSO, POS, OSP, OPS) are updated similarly for the given triple $s_1 p_1 o_1$. This schema is not efficient in terms of storage space, since each subject, predicate and object is stored six times over all tables. Additionally, all occurrences of the same resource URI or literal are stored more than once across all tables and within a single table. However, a find operation does not require a lookup across multiple tables and can be constrained.
Figure 6.6. Vertically Partitioned and Indexed Layout – Storage Schema
to the table that best matches the given search pattern. Moreover, the efficiency of “find” in this layout vs. the previous layouts depends on the performance contrast between row comparison vs. column retrieval and comparison in HBase. This is because in this layout we simply store triples as row keys while in previous layouts we used columns to store triples.

Vertically Partitioned and Indexed Layout

Figure 6.6 presents the storage schema for the Vertically Partitioned (VP) and Indexed layout. This layout is a combination of the previously described VP and Indexed layouts. The idea is to combine the advantages of both layouts while avoiding a considerable loss in
performance. As before, the VP part creates two tables for each unique predicate in a RDF graph, each of these tables is indexed by subjects and objects respectively. For the Indexed part, three tables are created in contrast with six tables that were created earlier. The SPO and OSP tables are retained, since they provide a complete set of triples for a given subject or object. In this way, multiple tables do not need to be searched for all triples associated with a subject or object, as was the case in the VP layout. In addition to the SPO and OSP tables, a table OS is created to store mappings between subjects and objects. A given object-subject pair is added as a row key without adding any new columns to the “blank” column family. This table is used to perform a lookup to ensure that a subject-object pair exists instead of exhaustively searching through the SPO and OSP tables. An add operation in this layout is a combination of addition operations in the VP and Indexed layouts. The difference is that a triple $s_1 p_1 o_1$ is only added to the SPO and OSP tables and a mapping $o_1 s_1$ is introduced in the OS table. In terms of storage space, this schema is neither as efficient as the VP layout, nor as inefficient as the Indexed layout. The overall efficiency in evaluating a find operation again depends on the efficiency of HBase to perform row comparison vs. column retrieval and comparison. However, for operations that involve subject- or object-based lookups, there could be an improved performance over the VP layout, since the SPO, OSP and OS tables can be used to perform such operations.

**Hybrid Layout**

Figure 6.7 presents the storage schema for the Hybrid layout, which is a combination of the Simple and Vertically Partitioned (VP) layouts. Again, the idea is to combine the advantages of these layouts without causing a drastic effect on performance. The predicates table from the Simple layout is not used, since the VP part creates separate tables for every unique predicate in an RDF graph. As before, the subjects and objects table of the Simple layout are retained, since they provide all triples belonging to a subject or object. These tables allow
Figure 6.7. Hybrid Layout – Storage Schema

us to avoid searching multiple tables for a given subject or object. An add operation in this layout uses addition operations for underlying Simple and VP layouts that were described earlier. Note that, for the Simple part, a triple needs to be added only to the subjects and objects tables since no predicates table exists. The storage space efficiency of this schema is neither as good as the VP layout, nor as bad as the Simple layout. The efficiency of evaluating a find operation in this layout should be better than other layouts, since we use columns across all tables and we also have separate tables for subjects and objects.

Hash Layout

Figure 6.8 presents the storage schema for the Hash layout. The Hash layout uses the exact same structure as the Hybrid layout. However, instead of storing nodes as strings in all tables, we store a hash of nodes in tables. The hash of a node is computed as an 8-byte integer
using the method “NodeLayout2.hash( Node )” from Jena SDB. Additionally, a separate table (name-pre-nodes) maintains a mapping between the hash of a node and its actual string representation. We used the Hybrid layout as the basis for the Hash layout, since it gives us the best performance out of all layouts, as will be evidenced by our experiments. The addition of a new triple to this layout is a two-step process: (1) Compute hash of nodes of a triple and then update all tables using hash values. (2) Create mappings between hash values and actual string representations of nodes. The storage space efficiency of this layout could be better than all other layouts since an 8-byte integer representation of nodes is used rather than their actual string representations.
6.2.3 Query Processing in Jena-HBase

There are two forms of query processing in Jena. A “find” operation returns all triples that match a pattern having form \(<S, P, O>\), where each component is a concrete value or a wildcard (*). A concrete value denotes that a node could be a URI, literal or blank node. A SPARQL query is converted into a series of find operations that may include variables to denote joins between triple patterns of a query. A “find” operation returns a set of bindings of the variables over triples in a graph. Each storage layout presents a different set of challenges for these operations. We discuss each operation for every storage layout below.

“Find” operation processing

Simple Layout: In this layout, we have three distinct tables each indexed by subjects, predicates and objects for every RDF graph. There are several cases to consider for “find” operation processing depending on the type of nodes in an \(<S, P, O>\) pattern.

- **A single node from \(<S, P, O>\) is concrete:** The concrete node is used to index into the appropriate table (\(S \rightarrow \) subjects table, \(P \rightarrow \) predicates table and \(O \rightarrow \) objects table) and the corresponding row of column names is returned. Each column name is matched with nodes of \(<S, P, O>\) other than the indexing node. All matching nodes are unified with the indexing node to return triples that match the \(<S, P, O>\) pattern.

- **Multiple nodes from \(<S, P, O>\) are concrete:** The following procedure is used to select an indexing node and its associated table to execute a find operation in this scenario: (1) The subjects table is tested if \(S\) is concrete. (2) The objects table is tested if \(O\) is concrete. (3) Finally, the predicates table is tested if \(P\) is concrete. This order is due to the reason that the subjects and objects tables lead to a better selectivity than the predicates table. Once the appropriate indexing node and table are obtained using the above process, a procedure similar to the one given in the bullet point above can be used to return triples that match an \(<S, P, O>\) pattern.
• All nodes from \(<S, P, O>\) are wildcards: An iterator on rows of the subjects table is constructed. For each iteration of the iterator, a new row from the subjects table will be used for obtaining triples. The number of triples obtained from a single row is equal to the number of distinct column names contained in that row. A single triple is constructed using each distinct column name along with the indexing subject node.

Vertically Partitioned Layout: This layout contains two tables, indexed by subjects and objects, for every unique predicate. Therefore, there are multiple triple tables for a graph. We present several scenarios depending on the type of node in an \(<S, P, O>\) pattern.

• A single node from \(<S, P, O>\) is concrete: This case is further divided into the following strategies based on which specific node of \(<S, P, O>\) is concrete.

  – \(S\) is concrete: For this scenario, all tables indexed by a subject (name-pre-\(p_i\)-subjects) are used. Since \(P\) is a wildcard in this case, \(<S, P, O>\) is passed, in turn, to each table for evaluation, until all tables are processed. The results obtained from all tables are returned to a user application in the form of an iterator.

  – \(O\) is concrete: This scenario is evaluated similar to the previous case except that, instead of using tables indexed by a subject, we now use tables indexed by an object (name-pre-\(p_i\)-objects).

  – \(P\) is concrete: In this case, the \(<S, P, O>\) pattern is processed by the particular subject based table containing the given predicate \(P\) in its table name.

• Multiple nodes from \(<S, P, O>\) are concrete: The same procedure that was used to select an indexing node and table in the Simple layout, is also used in this case to select whether to use subject based or object based tables. Additionally, if the predicate \(P\) is concrete, then, the \(<S, P, O>\) pattern is processed by the single subject or object table containing \(P\). On the other hand, if \(P\) is a wildcard, then the \(<S, P, O>\) pattern is passed, in turn, to each of the selected subject or object tables for evaluation.
- All nodes from $<S, P, O>$ are wildcards: In this case, all tables indexed by a subject (name-pre-$p_i$-subjects) are used. An iterator over all subject based tables is created. Additionally, each table will have a separate iterator over all rows of that table. For each iteration of this row iterator, a new row from that subject based table will be used for obtaining triples. The number of triples obtained from a single row is equal to the number of distinct column names contained in that row. A single triple is constructed using each distinct column name along with the indexing subject node and the predicate name which is part of the table name.

Indexed Layout: This layout creates six tables, one for each of the six possible combinations of nodes in a triple. The six tables give us greater flexibility in processing different scenarios of a “find” operation. These scenarios arise from different combinations of nodes being concrete in an $<S, P, O>$ pattern. We now present how these different scenarios are handled.

- A single node from $<S, P, O>$ is concrete: The concrete node’s value is used to index into a specific table ($S \rightarrow SPO$ table, $P \rightarrow PSO$ table and $O \rightarrow OPS$ table) and return only a subset of rows that match the given value. Then, for every row returned by the previous step, a new triple is constructed. Note that, triple construction requires reordering of nodes for a concrete predicate ($PSO \rightarrow SPO$) and object ($OPS \rightarrow SPO$).

- Multiple nodes from $<S, P, O>$ are concrete: The procedure described for the Simple layout is repeated to first select a node from $<S, P, O>$ that will be used as the indexing node. For example, if $S$ were concrete, irrespective of the other node types, $P$ and $O$, we would select $S$ to be the indexing node. The choice of table to process an $<S, P, O>$ pattern now additionally depends on which combination of nodes is concrete. For example, if $S$ and $P$ were concrete, we would use the SPO table while if $S$ and $O$ were concrete, we would use the SOP table. Again, only a subset of rows that matches an $<S, P, O>$ pattern is returned. Now, for every returned row a new triple would be
constructed. Note that, node reordering is required when using predicate based (tables PSO and POS) and object based (tables OSP and OPS) tables.

- All nodes from <S, P, O> are wildcards: An iterator over all rows of the SPO table is constructed. For each iteration of the iterator, a new triple is constructed (using the SPO row key) and returned to the user application.

Vertically Partitioned and Indexed Layout: This layout combines the Vertically Partitioned (VP) layout with a part of the Indexed layout. The idea is to use the Indexed part for “find” operations in which nodes, S and/or O in a given <S, P, O> pattern are concrete, thus avoiding searching over multiple tables as was the case with the VP layout. We present below how various scenarios based on varying node types are handled in this layout.

- A single node from <S, P, O> is concrete: This case is further sub-divided into different cases depending on which specific node in an <S, P, O> pattern is concrete.
  
  - S is concrete: In this case, we use the SPO table that is available to us. The concrete value for S is used to return only a subset of rows that match that value and then, a new triple is constructed for every matching row.
  
  - O is concrete: This case is similar to the previous case, except that we now use the OSP table to process the <S, P, O> pattern.
  
  - P is concrete: In this case, the <S, P, O> pattern is processed by the particular subject based table containing the given predicate P in its table name.

- Multiple nodes from <S, P, O> are concrete: The procedure described for the Simple layout is used to determine the indexing node. Additionally, this procedure determines if we should use subject or object based tables for the VP part. If the predicate P is concrete, we use the appropriate subject or object based table. On the other hand, if P is a wildcard, then we need to iterate over all subject or object based tables to ensure
that we obtain a complete result. Before we iterate over all tables, we use the OS table to ensure that a given subject-object pair exists. Finally, when all nodes $S$, $P$, and $O$ in an $<S, P, O>$ pattern are concrete, we use the SPO table to process the pattern.

- **All nodes from $<S, P, O>$ are wildcards:** In this case, an iterator over all rows of the SPO table is constructed. For each iteration of the iterator, a new triple is constructed (using the SPO row key) and returned to the user application.

**Hybrid Layout:** This layout combines a part of the Simple layout with the Vertically Partitioned (VP) layout. As before, the idea is to use the subjects and objects tables for “find” operations involving concrete $S$ and/or $O$ nodes. We now present how various scenarios based on varying node types in an $<S, P, O>$ pattern are handled in this layout.

- **A single node from $<S, P, O>$ is concrete:** This case is further sub-divided into different cases depending on which specific node in an $<S, P, O>$ pattern is concrete.
  - $S$ is concrete: In this case, we use the subjects table from the Simple part. The concrete value for $S$ is used to index into the subjects table and the corresponding row of column names is returned. A new triple is constructed for every unique column name along with the indexing subject node.
  - $O$ is concrete: This case is similar to the previous case, except that we now use the objects table to process the $<S, P, O>$ pattern.
  - $P$ is concrete: In this case, the $<S, P, O>$ pattern is processed by the particular subject based table containing the given predicate $P$ in its table name.

- **Multiple nodes from $<S, P, O>$ are concrete:** We again use the procedure described for the Simple layout to determine the indexing node. The same procedure is also used to determine whether we should use subject or object based tables for the VP part. If the predicate $P$ in an $<S, P, O>$ pattern is concrete, we use the appropriate subject or
object based table. On the other hand, if P is a wildcard, then we use the subjects or objects table to process an <S, P, O> pattern. Finally, for the case in which all nodes S, P, and O in an <S, P, O> pattern are concrete, we use the subjects table to process the pattern.

- All nodes from <S, P, O> are wildcards: In this case, an iterator over all rows of the subjects table is constructed. For each iteration of the iterator, a new row from the table will be used for obtaining triples. The number of triples obtained from a row is equal to the number of distinct column names contained in that row. A single triple is constructed using each column name along with the indexing subject node.

**Hash Layout:** This layout uses the same structure as the Hybrid layout. However, instead of storing string representations of nodes in all tables, it stores 8-byte integer representations of nodes in tables. A separate table maintains the mapping between a hash of a node and its actual string representation. In this layout, “find” operations are processed in a similar fashion to the Hybrid layout, since both these layouts share the same structure. Therefore, we do not present how these operations are processed, since this was given previously. However, we present two important differences that characterize “find” operation processing in this layout when compared with the Hybrid layout.

- The concrete nodes in an <S, P, O> pattern are first converted into their hash representations. Then, this converted triple pattern is processed depending on the various scenarios that were previously outlined.

- The triples that match a transformed triple pattern are converted back to their string representations before being returned to an application. This conversion is accomplished using the special mapping table (name-pre-nodes).
**SPARQL query processing**

In Jena, a SPARQL query is converted into a series of find operations that use variables to denote joins between generated triple patterns. The query is then processed as a nested loop through the find operations by using the result of one operation to bind values to variables, after which the subsequent operation’s pattern is evaluated. Depending on the layout used, a find operation is processed using the strategy given above. In the future, we plan to explore if a join in a SPARQL query can be pushed into the HBase processing engine, since this would lead to a more efficient query evaluation plan.

### 6.3 Performance Evaluation

This section presents details of our experimental investigation into the performance of Jena-HBase. We begin by presenting details of the experimental setup, followed by a brief description of benchmarks used in the experiments. We then present our findings that compare the various layouts using queries of the selected benchmarks. Finally, we also present a performance comparison of Jena-HBase with Jena TDB, which is a single-machine RDF storage framework, using the same benchmark queries.

#### 6.3.1 Experimental Setup

We conducted our experiments on a cluster containing 14 nodes. Each machine in the cluster consists of a Pentium IV processor with 290GB to 360GB disk space and 4GB main memory. The cluster ran Hadoop v0.20.2 and HBase v0.90.1. Additionally, the nodes are connected with a 48-port Cisco switch on an internally created private network.

Since we compared the performance of Jena-HBase with Jena TDB, we describe the configuration of the machine on which experiments were conducted for TDB. The machine consists of an Intel Core2 Duo E8400 3GHz CPU with a 250GB hard drive and 4GB main memory. The experiments used JRE v1.6.0_03 as the Java engine and Jena TDB v0.8.10.
6.3.2 Experimental Benchmarks

The experiments used the SP²Bench (Schmidt et al., 2009) and LUBM (Guo et al., 2005) benchmarks to evaluate the effectiveness of various Jena-HBase layouts and to compare the performance of Jena-HBase with Jena TDB. SP²Bench has been set in the DBLP (Ley and Reuther, 2006) library scenario, and consists of both, a data generator for creating arbitrarily large DBLP-like documents, and a set of benchmark queries. On the other hand, LUBM consists of a realistic university ontology, a data generator for generating OWL data sets that can be scaled to arbitrary sizes, and a set of inference queries.

6.3.3 Experimental Procedure

We performed two experiments with Jena-HBase. The first compared various Jena-HBase layouts with the explicit goal of determining the best layout, while the second compared the performance of the best layout with Jena TDB. We compared Jena-HBase only with Jena TDB and not with other Hadoop-based systems for the following reasons: (1) Jena TDB is the best in terms of query performance out of all available Jena storage subsystems. (2) The available Hadoop-based systems only implement some features proposed in the RDF specification. Additionally, we wanted to move away from the tedious task of writing MapReduce programs while providing application programmers with a system that provides all features mandated by the RDF specification.

We conducted all experiments using only those benchmark queries for which the running times were less than 10 minutes using Jena TDB for a graph size of \( \approx 1 \text{M} \) triples for SP²Bench and \( \approx 560\text{K} \) triples (5 universities) for LUBM. As a result, queries Q4, Q5a, Q6 and Q7 from Sp²Bench and queries Q2, Q7, Q8 and Q9 from LUBM were not used in any experiment, and therefore, results for these queries are not presented. Additionally, as a part of the procedure to determine the best layout, we ran both benchmarks over several graph sizes, but we show results only for a graph of 250137 triples for SP²Bench (Figures 6.9 and 6.10) and \( \approx 560\text{K} \)
triples (5 universities) for LUBM (Figures 6.11 and 6.12). Although we used a small graph size (250137 or ≈ 560K triples), it is still sufficient for the purpose of determining the best Jena-HBase layout. Furthermore, since the LUBM benchmark contains inference queries, we used the Pellet reasoner (v2.3.0) to perform inference tasks. Finally, in all subsequent graphs, an S denotes an HBase scanner timeout exception, while an M denotes an Out Of Memory exception. The HBase scanner timeout was set to 30 seconds (default value) and denotes the maximum allowable time between two invocations of a row scanner for an HBase table. Additionally, for experiments that compare the scalability of Jena-HBase with Jena TDB, a T denotes an Execution timeout, which is caused when a query takes very long to execute and is set to 60000 seconds.

6.3.4 Comparison of Jena-HBase layouts

In this subsection, we present a performance comparison of various Jena-HBase layouts using the SP²Bench and LUBM benchmarks. The goal of these experiments was to find the best layout from amongst the six available Jena-HBase layouts.

Comparison of Jena-HBase layouts for SP²Bench

Aim: The goal of this set of experiments was to determine the best layout from amongst the six layouts that are currently available in Jena-HBase using the SP²Bench benchmark.

Procedure: In these experiments, we executed different queries of the SP²Bench benchmark (queries other than Q4, Q5a, Q6, Q7) on a graph size of 250137 triples for various Jena-HBase layouts. Then, we analyzed the layouts across the following two metrics:

1. Graph Loading: We measured the time/space required to load/store the RDF graph consisting of 250137 triples for each Jena-HBase layout. Since HBase requires the periodic use of a “commit” operation to reflect changes to a table, a “commit” was performed once on all tables in a layout for every 20,000 triples added. Note that, the
value 20,000 was randomly selected and in the future we plan to carry out a separate analysis of the impact of this value on the overall graph loading time.

2. Graph Querying: We also measured the time required to execute different SP²Bench queries for each Jena-HBase layout.

Observations: We now present our observations for the two metrics outlined above.

1. Graph Loading: Figure 6.9 presents results of the graph loading experiment for various Jena-HBase layouts. We observe that the Indexed layout takes the least amount of time to store the graph of 250137 triples. This is because the Indexed layout only uses rows in HBase tables to store triples, while avoiding the time-consuming task of adding columns to tables like other layouts. However, since the Indexed layout needs to add triples to six tables, it also consumes the largest amount of storage space. If we consider the experiment purely from the perspective of storage space, we observe that the Vertically Partitioned layout takes the least amount of storage space. This is because the Vertically Partitioned layout does not store predicate information in any HBase table, the predicate URI is stored as part of the table name.

2. Graph Querying: Figure 6.10 presents results of the graph querying experiment for various Jena-HBase layouts. We observe that the Hybrid layout gives the best performance from amongst all layouts. This is because the Hybrid layout combines the
Figure 6.10. Comparison of Jena-HBase layouts for SP²Bench – Graph Querying (S denotes an HBase scanner timeout)
benefits of both, the Simple (e.g. see Q9) and Vertically Partitioned (e.g. see Q1) layouts. The remaining layouts all have disadvantages in at least one SP²Bench query. For example, the Simple layout always needs to scan the larger “predicates” table for graph patterns involving a concrete predicate and a wildcard for the subject and object. Therefore, for queries that contain such graph patterns, the Simple layout performs poorly when compared with, for example, the Vertically Partitioned (VP) layout (see queries Q2, Q5b, etc.) that only needs to lookup smaller “predicate” tables for the specific predicates present in the queries. However, the VP layout needs to scan multiple “predicate” tables for graph patterns that involve a concrete subject or object to ensure that a complete result is obtained. Therefore, this layout performs poorly for queries that contain such graph patterns when compared with the Simple layout (see query Q9) that has an aggregated view of all triples for a given subject or object in the “subjects” and “objects” table. The Indexed layout requires a longer querying time since it requires multiple row lookups in different HBase tables corresponding to an incoming graph pattern matching request. The VP-Indexed and Hash layouts also give querying times that are slightly longer than their corresponding base layouts (viz. VP and Hybrid), since for all queries they are required to perform row lookups on the SPO, OSP and OS tables (VP-Indexed case) or the mapping table (Hash case).

Comparison of Jena-HBase layouts for LUBM

Aim: The goal of this set of experiments was to determine the best layout from amongst the six available Jena-HBase layouts using the LUBM benchmark.

Procedure: In these experiments, we executed different queries of LUBM (queries other than Q2, Q7, Q8, Q9) on a graph size of $\approx 560K$ triples (5 universities) for various Jena-HBase layouts. Then, we analyzed the layouts across the following two metrics:
1. Graph Loading: Since the LUBM benchmark contains inference queries, the process of graph loading entails the following sub-tasks: (1) Triple Loading: This sub-task loads OWL data files into a layout. (2) Graph Reasoning: This sub-task builds the necessary data structures for the Pellet reasoner, which are subsequently used during the execution of LUBM inference queries. In the current set of experiments, we measured the time required to perform each of the previous sub-tasks for every layout during the loading of the RDF graph comprising \( \approx 560K \) triples. In addition, we measured the space required to store the same graph. As before, a “commit” operation was performed once on all tables in a layout for every 20,000 triples added to a graph.

2. Graph Querying: We also measured the time required to execute different LUBM queries for each Jena-HBase layout.

Observations: We now present our observations for the two metrics outlined above.

1. Graph Loading: Figure 6.11 presents results of the graph loading experiment for various Jena-HBase layouts. Firstly, we note that the Graph Reasoning time for the Indexed layout has been scaled down by a factor of 100 for a clear comparison with other layouts. As before, we observe that the Indexed layout takes the least amount of time to store the RDF graph comprising \( \approx 560K \) triples. Again, we attribute this to the fact that the Indexed layout only uses rows in HBase tables to store triples, while avoiding the
time-consuming task of adding columns to tables. However, similar to the observations made for the SP²Bench experiments, the Indexed layout consumes the largest amount of storage space from amongst the various layouts, since it needs to add triples to six tables. If we consider the experiment purely from the perspective of storage space, we again observe that the Vertically Partitioned layout takes the least amount of storage space. Again, this is because the Vertically Partitioned layout does not need to store predicate information in any HBase table, which is stored as a part of a table’s name. Figure [6.11] also shows that the Hybrid layout requires the least amount of time to perform the Graph Reasoning sub-task from amongst the various Jena-HBase layouts. This is because the Hybrid layout is the best available layout for querying purposes (from SP²Bench experiments), and since the Graph Reasoning sub-task internally uses a series of queries to the underlying layout for the purposes of building the necessary data structures, it is natural that the Hybrid layout outperforms the other layouts.

2. Graph Querying: Figure [6.12] presents results of the graph querying experiment for various Jena-HBase layouts. We note that the execution times for the Indexed layout have been scaled down by a factor of 50 for Q4, by 7 for Q12 and by 8 for Q14 for a clear comparison. Similarly, the execution time for the Hash layout has been scaled down by a factor of 20 for Q14 for a clear comparison with other layouts. From Figure [6.12] we observe that the Hybrid layout gives the best performance from amongst all layouts for LUBM queries. Additionally, we observe that for most of the benchmark queries, the Vertically Partitioned and VP-Indexed layouts also give query execution times that are very similar to the execution times for the Hybrid layout. This is because all three layouts are based on a vertical partitioning strategy that creates several small tables based on the predicates in a RDF graph rather than long, narrow tables created by the Simple and Indexed layouts. Furthermore, since LUBM queries always specify a concrete predicate in each triple pattern, the three layouts perform query execution over
Figure 6.12. Comparison of Jena-HBase layouts for LUBM – Graph Querying
smaller “predicate” tables, thereby leading to a lower query execution time. The use of “subjects” and “objects” tables, which provide an aggregated view of all triples for a given subject or object, allow the Hybrid layout to outperform the Vertically Partitioned and VP-Indexed layouts for queries containing a concrete subject or object (e.g. Q10 and Q13). As observed for experiments involving SP²Bench, the Simple layout needs to scan the large “predicates” table for graph patterns involving a concrete predicate and a wildcard for the subject and object. Therefore, for LUBM queries that contain such graph patterns, the Simple layout performs poorly when compared with other layouts (see queries Q4 and Q12). Also, the Indexed layout requires a longer querying time since it requires multiple row lookups in different HBase tables corresponding to an incoming graph pattern matching request. Finally, the Hash layout gives querying times that are slightly longer than its corresponding base layout (viz. Hybrid), since for all queries it is required to perform row lookups on the mapping table.

6.3.5 Comparison of Jena TDB vs. Jena-HBase

In this subsection, we present a performance comparison of the best Jena-HBase layout, viz. Hybrid, with a single-machine RDF storage framework, viz. Jena TDB using the SP²Bench and LUBM benchmarks. We specifically chose Jena TDB since it is the best performing model out of all available Jena models (In-memory, RDB, SDB and TDB).

Comparison of Jena TDB vs. Jena-HBase for SP²Bench

Aim: The goal of this set of experiments was to compare the performance of the Hybrid layout with Jena TDB for the SP²Bench benchmark.

Procedure: In these experiments, we executed different queries of the SP²Bench benchmark (queries other than Q4, Q5a, Q6, Q7) for increasing graph sizes (from ≈ 1M to ≈ 100M triples) on the Hybrid layout and Jena TDB. Then, we analyzed the performance of Jena-HBase vs. Jena TDB across the following two metrics:
1. Graph Loading: We measured the time to load various RDF graphs and the space required to store the graphs for the Hybrid layout and for Jena TDB.

2. Graph Querying: We measured the time required to execute different SP²Bench queries for the Hybrid layout and Jena TDB.

Observations: We now present our observations for the two metrics outlined above.

1. Graph Loading: Figure 6.13 presents results of the graph loading experiment for the Hybrid layout vs. Jena TDB. We note that the loading times for different graphs in Jena-HBase (the graph on the left hand side of Figure 6.13) have been scaled down by a factor of 15 for a clear comparison with Jena TDB. We observe that TDB outperforms the Hybrid layout in terms of both, the time and storage space required to store various RDF graphs. This is because although TDB stores a separate index for each of the six possible S-P-O combinations (similar to the Indexed layout), it uses separate nodes, prefixes and nodes-to-ids and prefixes-to-ids mappings to reduce the overall storage space. In addition, TDB makes use of a B+ tree structure that allows for fast lookups in the above mappings during graph loading (for accessing mappings for the same nodes or prefixes found in distinct triples), thereby reducing the overall storage time. On the other hand, the Hybrid layout stores URI’s, literals and blank nodes in a string representation, and moreover, stores the same nodes thrice across different HBase tables, thereby increasing the overall storage space. Additionally, the Hybrid layout requires a row and column lookup⁴ at the time of insertion of each new triple for every HBase table during graph loading, therefore increasing the storage time when compared with TDB. Finally, we also observe that TDB fails to construct a graph of ≈ 100M triples, since it runs out of memory when it tries to build the required graph indices, while the Hybrid layout is successfully able to construct this graph.

⁴A series of column lookups is required for every row lookup to determine the components of a triple.
2. Graph Querying: Figure 6.14 presents results of the graph querying experiment for the Hybrid layout vs. Jena TDB. We note that execution times for certain queries in Jena-HBase have been scaled down for a clear comparison with TDB. Specifically, the execution time of Q1 has been scaled down by 1000, Q3a by 5, Q3b by 150, Q3c by 100, Q8 by 1000, Q9 by 100, Q12a by 70000 and Q12b by 1000. We observe that TDB outperforms the Hybrid layout for all queries other than Q11 for the range between 1M - 25M triples. This is because for this range of triples, TDB is able to create and maintain the necessary indices (graph and mapping) in memory. Additionally, the use of a B+ tree structure results in an overall reduced, sometimes constant (viz. Q1, Q10, Q12a, Q12b, Q12c), lookup time in the graph indices and the mapping indices (node-to-ids and prefixes-to-ids) when compared with the Hybrid layout. On the other hand, the Hybrid layout makes multiple graph pattern matches on increasing graph sizes for SP²Bench queries, thereby requiring row and column lookups (see footnote 4) on successively larger HBase tables, resulting in a longer querying time when compared with TDB. Finally, we observe that for a graph size of \( \approx 100M \) triples, TDB is unable to execute any query, since it is unable to construct a graph at this size. On the other hand, the Hybrid layout is able to execute queries at this graph size, thereby confirming
Figure 6.14. Comparison of Jena TDB vs. Jena-HBase Hybrid layout for SP²Bench – Graph Querying (M denotes an Out Of Memory exception while T denotes an Execution timeout)
our intuition that Jena-HBase can be used to overcome the scalability issue faced by current single-machine RDF frameworks such as Jena TDB.

**Comparison of Jena TDB vs. Jena-HBase for LUBM**

**Aim:** The goal of this set of experiments was to compare the performance of the Hybrid layout with Jena TDB for the LUBM benchmark.

**Procedure:** In these experiments, we executed various queries of LUBM (queries other than Q2, Q7, Q8, Q9) for increasing graph sizes (from 50 universities - \( \approx 5 \text{M triples} \) to 1000 universities - \( \approx 100 \text{M triples} \)) on the Hybrid layout and Jena TDB. Then, we analyzed the performance of Jena-HBase vs. Jena TDB across the following two metrics:

1. **Graph Loading:** As stated earlier, the process of graph loading entails sub-tasks that perform Triple Loading and Graph Reasoning. Therefore, in this set of experiments we measured the time required to perform each sub-task for the Hybrid layout and Jena TDB as the graph size increases. In addition, we also measured the space required to store these graphs for both, the Hybrid layout and Jena TDB.

2. **Graph Querying:** We also measured the time required to execute different LUBM queries for the Hybrid layout and Jena TDB.

**Observations:** We now present our observations for the two metrics outlined above.

1. **Graph Loading:** Figure 6.15 presents results of the graph loading experiment for the Hybrid layout vs. Jena TDB. We note that loading times for various graphs in Jena-HBase (the leftmost graph in Figure 6.15) have been scaled down by a factor of 8 for a clear comparison with Jena TDB. We observe that TDB outperforms the Hybrid layout in terms of both, the loading time and storage space required by various RDF graphs. As observed in the case of SP²Bench experiments, this reduction in loading time and
storage space is due to TDB’s use of graph and mapping indices. On the other hand, the Hybrid layout stores each triple multiple times in its original representation, thus increasing the overall storage space. Furthermore, the Hybrid layout also requires row and column lookups (see footnote 4) whenever a new triple is inserted into an HBase table, thereby increasing the loading time. We also observe that Jena TDB fails to construct a graph for 1000 universities (∼100M triples), since it runs out of memory when it tries to build the required graph indices, while the Hybrid layout is successfully able to construct such a graph. Figure 6.15 also shows that Jena TDB fails to perform the Graph Reasoning sub-task for LUBM datasets in the range 50-1000 universities, thereby preventing the execution of LUBM inference queries (viz. Q3, Q4, Q5, Q6, Q10, Q11, Q12 and Q13) in this range. This is because the test program runs out of memory when Pellet tries to build inference related data structures by querying the underlying graph in TDB, which returns results that cause an overflow of memory. On the other hand, Pellet is successfully able to build necessary data structures by querying graphs stored in the Hybrid layout for the same range of universities.

2. Graph Querying: Figure 6.16 presents results of the graph querying experiment for the Hybrid layout vs. Jena TDB. We note that execution times for certain LUBM
Figure 6.16. Comparison of Jena TDB vs. Jena-HBase Hybrid layout for LUBM – Graph Querying (M denotes an Out Of Memory exception)
queries in Jena-HBase have been scaled down for a clear comparison with Jena TDB. Specifically, the execution times of queries Q1 and Q14 have been scaled down by a factor of 10 and 5. We observe that TDB outperforms the Hybrid layout for Q1 and Q14 (non-inference queries) for the range of universities between $N = 50$ to 500. As observed earlier, for these queries TDB can create and maintain the necessary B+ indices (graph and mapping) in memory, thus resulting in a reduced querying time. On the contrary, the Hybrid layout requires multiple lookups in successively larger HBase tables, thereby resulting in an increased querying time. From Figure 6.16, we also observe that TDB fails to execute inference queries in LUBM for the range between 50 to 500 universities, since the test program runs out of memory during the operation of the Graph Reasoning sub-task. On the other hand, the Hybrid layout successfully executes all queries for this range of universities, since Pellet is able to construct necessary inference related data structures. Finally, we observe that TDB fails to construct a graph for $N = 1000$, thereby preventing execution of all LUBM queries at this size. On the other hand, the Hybrid layout is able to execute all queries for $N = 1000$, thereby reaffirming our intuition about the scalability of Jena-HBase.
In Chapter 1, we outlined various challenges, namely cultural, policy, governance, economics and secure technological infrastructure, associated with Assured Information Sharing (AIS). In addition, we discussed how this dissertation attempts to address the lack of solutions for the secure technological infrastructure challenge. In this chapter, we survey the significant body of existing research work that has contributed towards the development of these solutions. The rest of this chapter is organized as follows: In Section 7.1 we survey existing policy-based AIS solutions that employ non-cloud data and policy engines to address the secure technological challenge. This work forms the basis of CAISS-X and CAISS, two novel private cloud-based solutions that address the secure technological challenge. Next, in Section 7.2 we outline existing work in the area of secure, distributed data processing in the context of cloud computing and databases. This work serves as the inspiration for Hybridizer, a framework that allows organizations participating in the AIS process to automatically partition their data and processing tasks over a hybrid cloud. After that, Section 7.3 summarizes existing work in the area of secure social network management in the context of Cloud Computing and Semantic Web technologies, which serves as the motivation for StormRider, an AIS underpinning that provides developers with tools that securely store, query and analyze large-scale, evolving networks in real-time. Finally, in Section 7.4 we outline the research work in the area of efficient storage and retrieval of RDF data. This work has influenced the development of Jena-HBase, a framework that uses existing Cloud Computing technologies to construct a distributed, scalable and efficient RDF storage framework, which is in turn used by StormRider to store/query network data in real-time using RDF/SPARQL.
7.1 CAISS-X and CAISS

In this section, we present a summary of existing policy-based AIS implementations that employ non-cloud data and policy engines.

**Policy-based Information Sharing:** A number of tools have been developed for allowing organizations to share information, while regulating access to shared data using policy-based engines. Although these tools offer discernible advantages over prior information sharing efforts, none of them are cloud-based and therefore, they lack the scalability, flexibility and efficiency needed for supporting a large user-base utilizing vast quantities of data.

In (Thuraisingham et al., 2008), the authors design and develop an AIS framework for information dissemination during emergencies using a Trusted Computing Base (TCB) that supports the “need to share” paradigm. The TCB is used to construct a hierarchy of system users based on their level of trust. Then, when an emergency occurs, the system disseminates information based on the severity of the emergency and the level of trust in a user. The framework from (Thuraisingham et al., 2008) is extended in (Awad et al., 2010) to facilitate information sharing using a trusted broker. In particular, the authors envision an AIS framework comprising the following steps: (1) Each organization defines domain-specific policies. (2) An organization partitions its dataset based on the data it wants to share using a horizontal, vertical or hybrid partitioning strategy. (3) When some analysis is to be performed, a trusted broker requests data from cooperating parties. (4) Every organization applies its policies on the partitioned dataset, which is subsequently given to the trusted broker. (5) The trusted broker performs the desired analysis using the shared, policy-compliant datasets.

In (Finin et al., 2009), the authors propose an information sharing life cycle (AISL) to realize the information sharing value chain of DoD (US-DoD, 2007a). AISL consists of three phases: (1) Information discovery and advertising. (2) Information acquisition, release and integration. (3) Information usage and control. Each phase entails several challenges such as, finding out what information can be shared and with whom, in phase (1), verifying if a
participating entity really needs the information that is being requested, in phase (2), and ensuring that information is used appropriately once it is released to a party, in phase (3).

The authors in (Xu et al., 2009) propose a framework called TIUPAM (Trustworthiness-centric Identity, Usage, Provenance, and Attack Management). The authors contend that with the shift towards the “need to share” paradigm, it is imperative to build a framework that can manage trustworthiness of information and risks associated with using/not using information. Towards this end, they propose a framework that includes the following components: (1) Identity management: Allow participating entities to verify the trustworthiness of shared information. (2) Usage management: Checks whether users are authorized to perform activities. (3) Provenance management: Used to verify the accuracy of information, data sources, people, etc. (4) Attack management: Handles procedures that need to be followed when the system is attacked or unauthorized activities are detected.

A separate work has focused on the development of a comprehensive “Environment for XACML policy Analysis and Management (EXAM)” (Rao et al., 2008; Lin et al., 2010) in the context of XACML policies. This work is motivated by the requirements of policy integration and interoperability that arise when organizations with widely varying access control policies want to collaborate. EXAM provides a variety of policy analysis techniques where each approach is implemented through analysis queries. An analysis query is a function that allows a user who has designed or deployed a policy to validate the accuracy of properties of that policy. Additionally, EXAM provides a number of analysis queries for the single as well as multiple policy scenarios. Furthermore, EXAM allows users to combine analysis queries to conduct arbitrarily complex analyses. EXAM also includes a policy similarity analyzer that implements the strategies necessary for the execution of analysis queries, and is based on the Multi-Terminal Binary Decision Diagram (MTBDD) (Fisler et al., 2005) and SAT-solver techniques (Fisler et al., 2005; Agrawal et al., 2005; Backes et al., 2004).
7.2 Hybridizer

In this section, we begin by discussing existing work done in the area of secure, distributed data processing in the context of cloud computing and databases.

Our work builds on previous research on data partitioning (e.g., Agrawal et al., 2004; Rao et al., 2002; Ghandeharizadeh and DeWitt, 1990; Curino et al., 2010), distributed query processing (e.g., evolution from systems such as SDD-1 (Jr. et al., 1980) to DISCO (Tomasic et al., 1996) that operates on heterogeneous data sources, to Internet-scale systems such as Astrolabe (van Renesse et al., 2003), and cloud systems (Logothetis and Yocum, 2008), and data privacy (Curino et al., 2011; Hacig¨ um¨ u¸ s et al., 2002; Gennaro et al., 2010).

Data and Workload Distribution over Multiple Clouds: There has been some recent work related to the problem of distributing workloads and data over a multi-cloud setup. In (Khadilkar et al., 2012), the authors present a formalization of the workload partitioning problem as a mechanism for maximizing workload performance. Additionally, the paper demonstrates the flexibility of the formalization by showing how existing systems such as (Zhang et al., 2011; Oktay et al., 2012) can be derived from the general workload partitioning framework. However, the paper simply proposes a preliminary vision for the workload distribution problem without providing any algorithmic solutions. In (Oktay et al., 2012), the authors address the problem of partitioning Hive workloads and the associated relational data over a Hybrid cloud model, while taking into account performance, monetary cost and data disclosure risk requirements. The paper formalizes the problem and provides a dynamic programming approach to solve the problem. A second paper, (Pattuk et al., 2013), addresses the problem of partitioning Key-Value data/workloads over a multi-cloud setup, while maximizing workload performance and ensuring that monetary cost and disclosure risk constraints are satisfied. Again, the paper presents a problem formalization along with a heuristic Hill-Climbing approach to solve the problem. The previous papers attempt
to formalize and solve the data/workload partitioning problem for a specific data/workload-cloud model combination. *Hybridizer* covers all Hybrid cloud models, irrespective of the data/workload model. Furthermore, we provide specific formulations of the problem for each of the Hybrid cloud models.

**Secure Distributed Data Processing:** Data partitioning has been studied extensively in distributed and parallel databases from a variety of perspectives, ranging from load balancing (Ghandeharizadeh and DeWitt, 1990), efficient transaction processing (Curino et al., 2010) to physical database design (Agrawal et al., 2004; Rao et al., 2002). In (Rao et al., 2002), the authors consider the problem of workload driven horizontal data partitioning for parallel databases. They propose a two step approach wherein the first step involves candidate partition generation and the second step is partition evaluation, which selects the best partition for each table with respect to the given workload. They generate partitioning “keys”, which are sets of attributes to be considered together as keys to hash functions that a partitioner will use. They use columns that appear in join and grouping statements to generate candidate keys. In contrast, we use predicates appearing in query statements (or predicates derived from them) as criteria for partitioning data across private/public clouds, thus giving us greater control over deciding which records get replicated on the public side, since disclosure risk is also a concern unlike their model. Furthermore, having a predicate based partitioning makes our statistics more accurate and efficient to compute than in (Rao et al., 2002) where they resort to sampling after the partitioning function is applied.

A more recent related paper (Curino et al., 2010) looks at the data partitioning problem in distributed databases for supporting OLTP queries efficiently. The objective is to improve throughput by reducing the time it takes to commit a transaction that needs to access records distributed across multiple nodes of a cluster with a shared nothing architecture. The time taken to complete a transaction is dependent on whether it accesses data on a single or multiple nodes and therefore, reducing the number of multi-node transactions can
significantly increase throughput. They propose a graph-based data partitioning (including limited replication) approach based on a well known class of graph partitioning algorithms called METIS \cite{Karypis:1995:GPG:223636.223640}, which are known to generate balanced partitions. The idea is to store all nodes within each partition at a single cluster node and minimize the number of edge crossings between different partitions, which in turn minimizes multi-node transactions. Previous approaches have also used round-robin, range and hash partitioning based techniques \cite{DeWitt:1992:DRD:133316.133343} for distributed and parallel databases. However, a graph partitioning based approach may not be very suitable for our setting, most importantly because of the poor scalability of such algorithms with size of a graph. Since graph size is proportional to the number of records in tables, such an approach is not amenable (as yet) for even medium sized databases. Also, since we are not considering typical OLTP workloads, the execution time of a query is not as dependent on it being single or multi site (which is the focus in \cite{Curino:2010:AMM:1863908.1863914}) as it is on the size of the intermediate and final result.

The work most related to ours is Relational Cloud, proposed in \cite{Curino:2011:EGR:2040283.2040295}, which addresses a similar hybrid cloud problem. Relational Cloud uses the graph-based partitioning scheme described above to split data into private and public sides. The partitions are encrypted with multiple layers of encryption and stored on a server. A query is executed on the encrypted data with multiple rounds of communication between a client and server without considering the cost of decrypting intermediate relations. The difference between our work and Relational Cloud is that our partitioning scheme explicitly considers the cost, both monetary and performance, of executing queries over data in private/public clouds.

A recent system, MONOMI \cite{Tu:2013:MCQ:2516590.2516602}, has been developed for executing analytical queries on encrypted data. The system first encrypts a database on a client, then ships the encrypted database to a service provider, and finally allows the execution of queries on the outsourced encrypted database. The creators of MONOMI have envisioned the following challenges, and solutions, for efficiently and securely executing analytical workloads over
encrypted data: First, I/O operations such as reading data from disk serve as the main bottleneck in executing queries over large datasets. To address this challenge, MONOMI proposes an intra-query split, viz., splitting client-server execution of queries that allows one to execute a query as much as possible over encrypted data on a server, and then executing the remaining parts on a trusted client, with decryption and post-processing as required. Second, processing analytical queries over encrypted data could be inefficient, since they contain complex computations. Moreover, it is challenging to partition a query into a server query that operates on the available encryption schemes on a server, and a client query. To address this challenge, MONOMI introduces specific optimizations, such as per-row precomputation, pre-filtering, etc., which aim to improve performance for certain types of queries. Third, since some queries may perform better than others, a physical layout design and planning phase is required for a given database/workload. To address this challenge, MONOMI introduces a designer that optimizes physical data layouts and a planner that decides how to split a query between the server and client, given a representative mix of analytical queries on a database. While MONOMI provides an interesting solution to the problem of executing queries over outsourced databases, we provide a general framework that intelligently partitions data/workloads, irrespective of their models (relational/Hive, RDF/SPARQL, etc.), under various cloud models. Furthermore, MONOMI chooses to encrypt the entire database, thus minimizing disclosure risk but potentially massively increasing monetary costs, whereas we take a more risk-based, cost-effective approach that selectively partitions data/workloads over a hybrid cloud.

Another relevant work, is a paper by Ko et al. (Ko et al., 2011) that proposes a “hybrid execution (HybrEx)” model that splits data and computation between public and private clouds for preserving privacy and confidentiality. Their focus, however, is on developing a distributed MapReduce framework that exploits both cloud infrastructures while maintaining privacy and confidentiality constraints. They do not deal with higher level query processing or optimization issues as we do in our work.
Chul Tak et al. (Tak et al., 2011) look at the economics of cloud computing and try to answer the question of “when does it make economic sense to migrate to the cloud?” They identify a set of factors that may affect the choice of deployment (in-house, cloud, and hybrid). They use the notion of “Net Present Value (NPV)” to calculate the profitability of an investment over its expected lifetime considering all cash inflows and outflows. Their main finding is that in-house provisioning is cost-effective for medium to large workloads, whereas cloud-based options suit small workloads. While this work is related in parts to our work, specifically to the monetary cost estimation, its focus is largely complementary to that of our work. However, we believe that more research needs to be done to conclusively decide whether supporting large workloads in a cloud model is beneficial under all circumstances.

A paper by Aggarwal et al. (Aggarwal et al., 2005) considered the problem of secure query processing using a distributed architecture. The authors propose a solution for secure outsourcing of relational data using two non-colluding servers. The privacy policies are stated in terms of subsets of attributes. The goal is to not give either server access to all attributes specified in a policy. The techniques employed are vertical partitioning of data along with selective encryption of some attributes. The data partitioning algorithm tries to partition attributes across the servers such that no set of attributes appears in plaintext on either server. While the two-server model can be mapped to our case, where the private cloud is both, a server and a trusted client issuing a query, their model of disclosure risk is completely different from ours. While they do not allow all attributes specified in a policy to be exposed to either server at any time, we are willing to do so in a controlled manner. This relaxation, makes our fundamental solution approach quite distinct from their approach.

Multi-objective Optimization Techniques: Given the metrics we consider in the formulation of the workload distribution problem ($WDP$), namely performance, data disclosure risk and resource allocation cost, it should be evident that one would like to maximize performance while minimizing the other metrics. Thus, one could formulate $WDP$ as a
multi-objective optimization problem for any Hybrid variant, however, given the conflicting nature of the metrics, it is difficult to do so in practice. Since a multi-objective optimization problem allows one to model a user’s preferences, there exist a multitude of methods to solve a problem based on how a user is allowed to specify their preferences \cite{Marler2004}. The first class of methods involves an *a priori articulation of preferences*, which allow users to specify the importance of objective functions before the optimization algorithm is executed. The second class of methods involves *a posteriori articulation of preferences*, in which a user’s preferences are used to select one solution from a set of mathematically equivalent solutions. The third class of methods do not account for a user’s preferences when solving an optimization problem. Finally, one could also use methods based on genetic algorithms to solve a multi-objective optimization problem \cite{Marler2004}.

In *Hybridizer*, we use the \( \varepsilon \)-constraint approach \cite{Haimes1981}, which is a special case of the bounded objective function method \cite{Hwang1979} from the *a priori* class of methods, which minimizes the single-most important objective function, while the remaining objective functions are used to form additional constraints \cite{Marler2004}. In *Hybridizer*, we minimize the most important objective function, for example, the overall workload performance, for a given deployment model, *viz.* Hybrid, and form constraints out of the remaining objective functions, *viz.* data disclosure risk and resource allocation cost. The reason for selecting the \( \varepsilon \)-constraint approach is the robustness with which it allows one to model an end-user’s preferences in the formulation of *WDP*. Furthermore, an end-user has sufficient background information on the cloud deployment model they want to use, as well as metric information such as overall budget, thus making the \( \varepsilon \)-constraint approach ideal for the formalization of *WDP*. Finally, the low computational complexity overhead and relative ease of developing algorithmic solutions also makes the \( \varepsilon \)-constraint approach particularly attractive for formulating *WDP* \cite{Marler2004}.

One may also wish to use other methods from the *a priori* class of methods such as weighted sum \cite{Zadeh1963} or exponential weighted criterion \cite{Athan1981}.
While these methods may be effective in certain situations such as when user preference information is limited or does not exist, they suffer from the drawback that despite having many effective techniques for finding coefficients, i.e., weights or other method-specific parameters, one may still be unable to find an assortment of weights that leads to a satisfactory solution (Marler and Arora, 2004). One may wish to explore methods from the a posteriori class of methods. However, since these methods involve some type of graphical or tabular presentation of optimal solutions to a user before a selection can be made, they do not scale to optimization problems involving a large number of objective functions. Although $WDP$ has a small number of objective functions, thus potentially allowing the use of a posteriori methods, the large computational overhead of providing a user with a complete Pareto optimal set makes the use of these methods prohibitively expensive (Marler and Arora, 2004). One may also wish to explore methods with no articulation of preferences, however, since a user typically has background knowledge such as the cloud deployment model to be used, the overall budget, etc., we do not explore this class of methods to formulate $WDP$. Finally, one could explore the use of genetic algorithms to directly solve $WDP$ in its original multi-objective form, however, their similarity with a posteriori methods makes them susceptible to the drawbacks of these methods, mainly the high overall computational expense, thus making them prohibitively expensive (Marler and Arora, 2004).

### 7.3 StormRider

In this section, we present a summary of existing work in the areas of storage, query and analysis of networks in the context of Cloud Computing and Semantic Web technologies. In addition, we survey existing approximation techniques for centrality estimation and social network access control techniques.

**Graph Databases:** A graph database uses the concept of a graph (viz. a set of nodes connected by edges) to represent and store data. Graph databases are also optimized for
answering graph-like queries. Since a number of graph databases have been developed by the
research community, in the subsequent discussion we limit our attention to graph databases
that use an RDF representation to store data. Although the frameworks described in the
discussion below use RDF as its data model for representing networks, they either suffer from
scalability issues (e.g. AllegroGraph) or are incapable of querying or analyzing networks as
they evolve (e.g. Bigdata and Virtuoso). These drawbacks do not allow the described
frameworks to handle interesting use cases that arise due to the evolving nature of networks.

AllegroGraph\(^1\) is a database and application framework for building Semantic Web appli-
cations. It allows storing data as RDF and querying RDF data through various query APIs
for SPARQL and Prolog. Additionally, AllegroGraph provides RDFS++ reasoning capabilities with its built-in reasoner. AllegroGraph also includes support for Federation, Social
Network Analysis, Geospatial capabilities and Temporal reasoning. The AllegroGraph SNA
tool provides a library that allows us to treat a RDF triple store as a social network with the
ability to calculate centrality, importance and different search functions. Examples of cen-
trality for actors and groups are degree, betweenness and closeness. Additional capabilities
provided with the SNA tool are techniques for finding cliques and shortest paths between
actors. The advantage of using this tool is that the representation of the social network is
in RDF, while the drawback is scalability, since this tool works on a single machine.

Bigdata\(^2\) provides a high performance RDF database capable of providing a fast load
throughput and best-in-class query performance. Additionally, Bigdata allows users to per-
form large-scale semantic alignment and data set federation. Further, the Bigdata RDF
database provides RDFS and limited OWL inference as well as full text search and indexing
using Lucene. Finally, the Bigdata RDF database also allows users to define custom rules
for conducting inference and performing analysis on networks stored in the database.

\(^1\)http://goo.gl/kN2CZM
\(^2\)http://goo.gl/Q28tMY
The Virtuoso Universal Server\(^3\) uses a hybrid server architecture to support functionalities of various data models such as relational, RDF, XML, etc. The Virtuoso RDF Quad-Store\(^4\) provides basic capabilities required of any RDF framework such as storage in several formats (HTML+RDFa, RDF-JSON, N3, etc.), SPARQL query processing and inference (using a backward chaining OWL reasoner). In addition, the store provides security features, indexing capabilities and SPARQL extensions for geo-spatial queries, full-text queries, etc.

**Social Network Analysis tools:** The rapid growth of web and online social graphs has resulted in the development of several frameworks that provide efficient strategies for analyzing graphs. We now provide a brief summary of the key features of some of these frameworks. Although the frameworks provide analysis and mining capabilities on large networks, they are unable to automatically perform these tasks on an evolving network. To perform this task, these frameworks require a user to manually enter different snapshots of a network, which are then analyzed to detect changes across them.

*General purpose social network analysis tools:* JUNG\(^5\) is a software library for modeling, analyzing and visualizing data that can be represented as a graph. Towards this end, JUNG provides support for various graph representations such as directed and undirected graphs, hypergraphs, etc. Additionally, JUNG includes implementations of various graph algorithms that perform clustering, importance measures, statistical analysis, etc. Finally, JUNG also provides tools to explore a graph using a built-in or custom designed visualization layout.

*ORA\(^6\)* is a dynamic meta-network assessment and analysis tool. It contains metrics for analyzing static and dynamic networks as well as techniques for comparing networks at various levels (network, group and individual). Additionally, *ORA can be used to examine

---

\(^3\)http://virtuoso.openlinksw.com/

\(^4\)http://virtuoso.openlinksw.com/rdf-quad-store/

\(^5\)http://jung.sourceforge.net/

\(^6\)http://goo.gl/ihlKgo
evolving networks through analysis of trail data; in this respect, *ORA is similar to Storm-Rider, which tries to detect changes in an evolving network. However, *ORA requires a user to manually enter snapshots of a network, which are analyzed to detect changes across them.

Etío et. al. provide formal definitions of SNA operators in SPARQL [Etienne et al., 2009]. They also propose SemSNA, an ontology that models SNA characteristics, which can be used to annotate social networks. However, rather than model SNA operators in SPARQL, we created topologies for them in StormRider, thus allowing us to analyze larger graphs than those in [Etienne et al., 2009] while maintaining a reasonable level of performance.

Cloud-based social network analysis tools: There are several cloud-based tools that provide SNA capabilities. X-RIME provides a MapReduce library that provides different analysis functions for finding strongly and weakly connected components, shortest paths, maximal cliques, pagerank, vertex degree statistics etc.

PEGASUS is a Peta-scale graph mining system [Kang et al., 2009] that uses MapReduce to compute degree distribution and pagerank, radius estimation, connected components, triangle counting and Random Walks with Restart (RWR). Again, the advantage of PEGASUS is its scalability, since it uses Hadoop. However, PEGASUS only provides analytical capabilities for a network, while ignoring aspects related with storage and retrieval of networks.

HaLoop [Bu et al., 2010] is a modified version of Hadoop that provides built-in support for iterative processing, which is an implicit part of graph processing. HaLoop achieves this goal by using caching mechanisms that make the task scheduler loop-aware. Additional features of HaLoop include: (1) Allowing users to reuse building blocks from standard Hadoop iterative application implementations. (2) Providing intra-job fault tolerance similar to Hadoop.

Pregel [Malewicz et al., 2010] and Apache Giraph provide cloud-based computational models for efficient processing of large graphs. A processing task is implemented as a sequence

http://xrime.sourceforge.net/
http://goo.gl/eqZVaf
of iterations, where in an iteration, a node can receive messages from a previous iteration and can send messages to other nodes. Additionally, a node can modify its state and that of its outgoing edges. Furthermore, the node-centric approach is general enough to support a several analysis algorithms. Finally, since these tools are cloud-based they provide several advantages such as efficiency, scalability and fault tolerance. The key difference between Pregel and Giraph is that the latter adds fault tolerance to processes that coordinate task iterations by using ZooKeeper as its centralized coordination service.

SNAP is a C++ framework for network analysis and graph mining. SNAP can generate and manipulate large graphs (regular and random), compute structural properties of a network such as number of triads, and support attributes for nodes and edges in a network.

GraphLab is a framework that compactly expresses asynchronous iterative machine learning algorithms with sparse computational dependencies, while ensuring data consistency and a high degree of parallel performance. GraphLab provides the following analog to Map and Reduce abstractions: (i) Map → Update Function: This function allows reading and modifying overlapping sets of data. Additionally, this function can be triggered recursively, thus allowing a chain of update functions to be applied to a graph. (ii) Reduce → Sync Operation: This function allows a reduce operation to be performed in the background, even when other computations are being performed. Also, this operation can read multiple records at the same time, thus enabling a task to execute on larger dependent contexts.

Cassovary is a “big graph” processing library designed to handle graphs with billions of nodes and edges in a space-efficient manner. Cassovary provides basic tools and algorithms necessary to analyze and mine large networks.

Approximation algorithms for centrality metrics: Several approaches for approximating centrality metrics, particularly closeness and betweenness, have been proposed by

---

9 http://snap.stanford.edu/snap/index.html
10 http://goo.gl/0HhtCd
the research community. These include sampling based techniques \cite{Eppstein2004,Okamoto2008,Bader2007}. The idea behind this technique is to repeatedly select nodes from a network based on a sampling technique (random, adaptive, etc.). Then, the required shortest path computations are performed using each sampled node as a starting point. The results of these calculations are then used to estimate centrality values for remaining nodes in the network. A parallel version of the algorithm given in \cite{Eppstein2004} is implemented in \cite{Bader2006}.

In \cite{Brandes2001}, Brandes presents an exact betweenness centrality calculation. The algorithm has the following steps: (1) Solve the single-source-shortest-path (SSSP) problem for each node. (2) After each iteration, dependencies of the source \((s)\) on every other node \((v)\) are added to the centrality score of that node, where a dependency of \(s\) on \(v\) is the fraction of all paths starting from \(s\) on which \(v\) lies. A parallel version of the exact algorithm is implemented in \cite{Bader2006}. An approximate version of the algorithm is constructed in \cite{Brandes2007} by extrapolating from a subset of \(k\) pivots but still using the aggregation strategy of the exact algorithm. In \cite{Geisberger2008}, the authors extend the work of \cite{Brandes2007} to improve betweenness for nodes that are overestimated due to being near pivot nodes. They achieve their goal by using an unbiased estimator and a scaling function.

The authors in \cite{Potamias2009} study approximate-landmark-based methods for path estimation in large networks. Their approach involves selecting a set of landmarks and computing the shortest distance from all other nodes to these nodes offline. Then, at runtime when the shortest distance for a pair of nodes is needed, it is estimated using precomputed distances. Additionally, the authors prove that the landmark selection problem is NP-Hard and they subsequently present different landmark-selection strategies such as Random, Degree and Centrality, along with constrained and partitioned variants of them.

In \cite{Rattigan2006}, the authors introduce the concept of a network structure index (NSI) that consists of a set of annotations for every node in the network, and a function
that uses annotations to estimate graph distances between pairs of nodes. Formally, for a graph \( G(V, E) \), annotations define a function \( A : V \rightarrow S \), where \( S \) is an arbitrarily complex annotation space, and the other element of the index is a distance measure \( D : S \times S \rightarrow \mathbb{R} \) that maps pairs of node annotations to a positive real number. NSI’s are based on different measures such as degree, landmark-based, etc. To estimate closeness centrality, they calculate the average of the graph distance to a sample of nodes in the data set. They use the NSI distance estimate to calculate the distance between the sampled pairs. To compute betweenness centrality, they randomly sample pairs of nodes and discover the shortest path between them and then count the number of times a given node appears on this path. They contend that with this method they can find somewhat accurate betweenness centralities with a much smaller proportion of nodes visited than with traditional breadth-first search.

The authors in [Santos et al., 2006] present a set of parallel algorithms to compute closeness and ego-centric betweenness centrality. They define an anytime anywhere methodology for this purpose. Anytime refers to providing results at any given time and refining the results over time. Anywhere refers to incorporating new information into the whole network. Their methodology consists of three parts: (1) Domain Decomposition: In this phase, the graph is partitioned into subgraphs such that cut size is minimized. (2) Initial Approximation: Any SNA tool is used to analyze each partition on a separate processor. (3) Recombination: This phase incrementally combines partial results to obtain the final results.

**Securing Social Networks:** Several social network access control techniques have been proposed. In [Carminati et al., 2011], the authors propose an extensible, fine-grained social network access control model based on Semantic Web technologies. Their idea is to capture social network information using an ontology in a Social Network Knowledge Base (SNKB), and access control policies defined as rules using an ontology in a Security Authorization Knowledge Base (SAKB). Then, access control is enforced by querying SAKB by means of a reasoner using instance checking operations or by executing a SPARQL query.
In (Kruk et al., 2006), the authors present D-FOAF, a FOAF-based distributed identity management system for social networks. This work demonstrates how information in a network can itself be used to provide community-driven access rights delegation. Furthermore, the paper also provides algorithms for managing distributed identity, authorizations and access rights checking.

In (Carminati et al., 2009), an access control mechanism for social networks is presented comprising policies defined as rules on resources, and authorized users defined in terms of type, depth and trust level of relationships between users. Furthermore, the work uses a semi-decentralized architecture, where access control is enforced on the client side by requiring a user to provide a proof that authorizes them to access the resource.

As stated earlier, all these works only ensure that authorized users are able to access network data, while neglecting issues related with storing and mining large-scale networks.

7.4 Jena-HBase

The area of efficient storage and retrieval of RDF data has been well studied by the research community. A lot of effort has been applied to develop solutions for efficient storage and retrieval of RDF data on single machines (Broekstra et al., 2002; Harris and Gibbins, 2003; Franz-Inc., 2005; McBride, 2002). However, as more and more data is expressed in RDF, single machine solutions tend to become inadequate. Therefore, to solve this data explosion problem it is necessary to develop multi-machine solutions, which automatically partition data into smaller subsets across machines and enable efficient processing of queries on distributed data. To this end, we present a short survey on such solutions in this section.

In (Castagna et al., 2009), the authors present theoretical designs of a parallel processing framework for RDF data. The key to such a framework is the strategy behind query execution, given a distribution of data over several machines. The data distribution could be achieved at varying levels of granularity such as triple, graph or dataset. At an abstract level,
any query could be rewritten into a set of sub-queries where each sub-query is evaluated on
the appropriate machine containing data associated with the sub-query. Then, results of all
sub-queries could be merged on a non-parallel component of the framework and can then be
returned to a user application. Several optimizations can be achieved in such a framework
based on the data distribution granularity as well as the query processing strategy.

Several research groups have built RDF storage and retrieval solutions using cloud com-
puting tools. In (Husain et al., 2009), the authors propose dividing a given dataset using
a vertically partitioned approach. The predicate-based datasets are then stored in Hadoop
Distributed File System (HDFS). A SPARQL query is executed in a series of iterative MapRe-
duce jobs, where a single iteration accomplishes a join of one variable in the query. There are
several challenges that need to be overcome by this work: (1) It does not provide any func-
tionality for a user to add new triples or delete existing triples from the added dataset. (2)
It only permits SPARQL queries that contain a basic graph pattern (BGP), and therefore,
does not support the entire SPARQL language. SPIDER (Choi et al., 2009) and $H_2RDF$
(Papailiou et al., 2012) are frameworks that distribute RDF data over several nodes in a
cluster using HBase tables. Then, traditional MapReduce processing is used to execute
queries over data. In contrast, our goal was to build a system that supports all features
mandated by RDF specification. We leave issues such as optimum data partitioning and
query optimization as a part of the future work. Similar efforts using Hadoop and HBase are
also given in (HEART, 2009; Abraham et al., 2010). HEART (HEART, 2009) is a project in
the Apache incubator that has not seen a lot of activity in the last few years. On the other
hand, the work in (Abraham et al., 2010) uses a layout that is similar to the Simple layout
that we have proposed. However, as our performance evaluation has shown, this schema is
very inefficient in terms of query processing when compared with the other layouts.

Another technique employed to store and retrieve RDF data uses peer-to-peer systems.
This technique has received significant attention by the research community (Cai and Frank,
A drawback of peer-to-peer systems is the need to have super peers that store information about the distribution of RDF data among peers in the network. Another disadvantage is a need to federate a SPARQL query to every peer in the network. Some systems such as Cai and Frank (2004) overcome these disadvantages by storing a triple once on three different peers where each peer is determined by a hash value of the subject, predicate and object of a triple. Then, at query time a node from the query can be hashed once more to determine the peer that stores the associated RDF data. This will significantly lower the query time when compared with query federation to all peers. A different approach to RDF storage and retrieval is taken in YARS2 (Harth et al., 2007) where six indices are created based on the different combinations of a triple in a named graph. Then, at query time the appropriate index can be used based on the incoming SPARQL query.

Several other proposals have been made for scalable RDF storage and querying. 4store (Harris et al., 2009) is a distributed RDF clustered store where each node stores a chunk of RDF data. The chunk to which a triple or quad belongs to is determined using an encoded form of its subject. For query processing, a central node aggregates all data associated with a query and then the query is executed centrally. A strategy based on a shared-nothing architecture is given in Weaver and Williams (2009). This work uses a hash join to map results of sub-queries to the same processor where a SPARQL join can be effectively performed. Similar research efforts have also been made towards inferencing over large scale distributed data (Urbani et al., 2010 and Weaver and Hendler, 2009) that use MapReduce to perform inference. Another approach that has been taken is Clustered TDB (Owens et al., 2008), which is a distributed storage subsystem for the Jena framework. This system takes the idea of TDB a step further by optimizing the storage of the node table and triple indices over a cluster. Query processing is performed by using designated query coordinators that receive a query, then convert a query into its canonical form and control the execution of a query over the data nodes.
CHAPTER 8
CONCLUSIONS AND FUTURE WORK

Assured Information Sharing (AIS) allows multiple cooperating organizations to securely share and analyze vast quantities of data. The process of information sharing, particularly in the context of the intelligence domain, has several real-world implications, the foremost of which is the ability to make proactive decisions that could help save thousands of lives. Furthermore, the information sharing process must ensure that sensitive information is safeguarded and the privacy of individuals is preserved. However, in today's precarious times, a lack of information sharing between organizations, particularly government agencies, could lead to incidents that cause irreparable damage to life and property. A relatively neglected area in the context of AIS is technological infrastructure, which refers to the existing IT infrastructure in an organization that allows it to efficiently, economically and securely share information. Although previous research has successfully developed policy-based information sharing systems, none of these operate in a cloud environment; therefore, they lack the scalability and efficiency needed for supporting a significant number of users utilizing vast quantities of data. In this dissertation, we attempted to remedy this situation by developing AIS implementations that operate under varying conditions.

The first part of this dissertation presented details of two prototypes that allow a large number of users to securely share/analyze large amounts of data. The first prototype, CAISS-X, used a cloud framework for managing relational data and a non-cloud policy engine to enforce XACML policies. Although CAISS-X represented a discernible enhancement over prior AIS implementations, it still suffered from constraints related with the usage of relational model and XACML-based policies. The second prototype, CAISS, overcame these limitations by using a cloud framework for data storage/retrieval and policy enforcement.
The next part of this dissertation presented *Hybridizer*, a framework that allows organizations to automatically partition their data/processing tasks over hybrid clouds, while taking into account performance, security and financial requirements. *Hybridizer* allows organizations to fully exploit the benefits of hybrid clouds towards developing AIS solutions, while achieving the right mix of performance, security and financial costs. We then empirically demonstrated that even under various deployment models, *Hybridizer* leads to considerable performance savings, a drastically reduced public cloud service acquisition cost and a reasonable sensitive data exposure risk, when compared with fully public cloud solutions.

The final part of this dissertation described StormRider, a system that allows organizations to securely manage large-scale, evolving networks in real-time. StormRider stored/queried network data using RDF/SPARQL due to their expressivity and ability to capture evolving domain requirements. StormRider also allowed one to perform a fundamental analysis operation, namely centrality estimation, on underlying evolving networks. Additionally, StormRider ensured that only authorized users gained access to data by employing various policy-based security mechanisms. We then empirically demonstrated that our adapted approximation algorithms for centrality estimation improve algorithmic performance by $\approx 25\text{-}45\%$, while maintaining a low approximation error of $\approx 3\%$. Furthermore, we showed that the overhead of enforcing redaction policies in StormRider is minimal (at most $5\%$). Since there did not exist a comprehensive, distributed RDF storage framework, the last part of this dissertation also described Jena-HBase, a framework that uses existing cloud technologies to construct a distributed, scalable and efficient RDF storage framework. We then empirically demonstrated that Jena-HBase provides enhanced scalability ($>25\%$) at the cost of additional overhead ($\approx 50\%$ in the worst case), when compared with Jena TDB.

In the future, we plan to explore the following areas of research:

- CAISS-X and CAISS: Incorporation of more diverse types of access control policies into CAISS-X and CAISS. These include, query rewriting based mechanisms, pattern matching based mechanisms, etc.
• Hybridizer: (1) Hybridizer currently solves the partitioning problem based on a given, user-defined workload. An interesting extension to Hybridizer would be to explore how to partition, or re-partition, data/processing tasks when we are given a continuously changing workload. (2) Currently, Hybridizer used the notion of inter-query parallelism to split data/workloads over hybrid clouds, however, we plan to analyze the effect of a more fine-grained approach, viz. intra-query parallelism, towards solving the partitioning problem.

• StormRider: (1) Implementation of algorithms for other SNA metrics such as diameter estimation, models of information flow, etc. (2) Currently, StormRider focused on storing Twitter data as RDF. However, as one of the design goals of the semantic web is data integration, we plan to investigate how data from multiple networks can be effectively integrated. This is a challenging task as it requires designing ontologies that allow data integration in the presence of ambiguities in data. (3) Incorporation of more expressive types of data security mechanisms in StormRider.

• Jena-HBase: Incorporation of a static query optimizer into Jena-HBase. This could be achieved in two steps. First, one is required to collect various statistics over a RDF graph as it is being constructed. The statistics need to be maintained at various levels. At the graph level, one is required to store statistics such as the number of triples in a graph, the number of unique subjects, etc. At the predicate level, one is required to store statistics such as the number of triples containing that predicate and the relative data distribution, which is stored as an equi-width or equi-height histogram. Second, one needs to implement a static query optimizer that decides the best execution plan for a given query. This task requires the implementation of an optimization algorithm, which in turn depends on selectivity estimation figures, which can be computed using the previously collected statistics.
In this appendix, we discuss the framework for performing RDF selectivity estimation, which is an extension of an existing framework [Stocker et al., 2008] that allows us to perform selectivity estimation for a wide variety of SPARQL operators.

A.1 Selectivity Estimation for RDF

In this section, we outline the framework for performing selectivity estimation for RDF in Hybridizer, which is based in part on the work given in [Stocker et al., 2008]. In particular, we discuss the approach used to collect statistics for RDF data, which are subsequently used to perform RDF selectivity estimation. We also discuss the techniques we have used to perform selectivity estimation for inference queries.

A.1.1 Statistics for RDF Data

In this subsection, we detail the statistics necessary to effectively perform RDF selectivity estimation. Since SPARQL graph patterns are typically composed of basic graph patterns, which are characterized by joins between triple patterns, we first outline the statistics required to perform selectivity estimation of triple patterns. Then, we provide a summary of statistics required to perform selectivity estimation of joined triple patterns. Finally, we describe how statistics collected for selectivity estimation of triple patterns/joined triple patterns can also be used for estimating selectivities of other SPARQL operators. A summary of statistics collected over all RDF/SPARQL constructs is given in Table A.1.
Table A.1. Summary of Statistics for RDF Data

<table>
<thead>
<tr>
<th>RDF/SPARQL Constructs</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Triple Pattern, TP</strong></td>
<td></td>
</tr>
<tr>
<td>Unbounded <em>sp</em></td>
<td>No statistics required</td>
</tr>
<tr>
<td>Bounded <em>sp</em></td>
<td>Total number of triples and total number of unique subjects</td>
</tr>
<tr>
<td>Unbounded <em>pp</em></td>
<td>No statistics required</td>
</tr>
<tr>
<td>Bounded <em>pp</em></td>
<td>The number of triples that match a given predicate</td>
</tr>
<tr>
<td>Unbounded <em>op</em></td>
<td>No statistics required</td>
</tr>
<tr>
<td>Bounded <em>op</em></td>
<td>Data type specific histograms for each distinct predicate</td>
</tr>
<tr>
<td><strong>Joined Triple Pattern</strong></td>
<td>- No. of triples matching predicates in the joined triple pattern</td>
</tr>
<tr>
<td></td>
<td>- Mappings in subject- and object-specific maps belonging to predicates</td>
</tr>
<tr>
<td><strong>FILTER</strong></td>
<td>Data type specific histograms over predicates</td>
</tr>
<tr>
<td><strong>Basic Graph Pattern, BGP</strong></td>
<td>All Previous</td>
</tr>
<tr>
<td><strong>UNION</strong></td>
<td>All Previous</td>
</tr>
<tr>
<td><strong>OPT</strong></td>
<td>All Previous</td>
</tr>
</tbody>
</table>
**Triple pattern statistics:** Since a triple pattern, $TP$, is composed of a subject, $sp$, predicate, $pp$, and object, $op$, pattern, the selectivity estimation of $TP$ requires one to collect specific statistics related with $sp$, $pp$ and $op$. Furthermore, since $sp$, $pp$ and $op$ could be bound or unbound, one also needs to take into account this dimension when performing selectivity estimation. The case when $sp$, $pp$ or $op$ is unbounded is trivial, since they have no impact on selectivity estimation, and therefore, statistics are not required to be collected for this case. On the other hand, the case when $sp$, $pp$ or $op$ is bound requires the collection of statistics to allow selectivity estimation. Furthermore, we use the size of a triple pattern, i.e., the number of triples that match a pattern, as a substitute for selectivity estimation, since one can easily compute the selectivity from the approximated size (Stocker et al., 2008).

The size of a *bound subject* is estimated as the average number of triples matching a subject. Therefore, the statistics required to determine the size in this case are the total number of triples and the total number of unique subjects in the graph. The size of a *bound predicate* is equal to the number of triples matching that predicate. Therefore, the exact statistic required to determine the size in this case is the number of triples that match a given predicate. The size of a *bound object* is estimated using equi-width histograms. In particular, a data type specific histogram is created for each distinct predicate to represent the distribution of object values. Note that, we can create data type specific histograms in our case, since we have *a priori* knowledge about the SP$^2$Bench schema.

**Joined triple pattern statistics:** In the case of joined triple patterns, we especially look at $TP$’s that are related because of a common variable between their subjects or objects. For instance, $TP$’s $[\text{?s} \ p_1 \ ?a]$ and $[\text{?s} \ p_2 \ ?b]$ are related because they share the common variable $\text{?s}$. Given a join relationship, *viz.*, subject-subject, subject-object, object-subject, object-object, between two related $TP$’s $tp_1$ and $tp_2$, the size of the corresponding joined triple pattern is used as an upper bound for the size of any $BGP$ involving $tp_1$ and $tp_2$. In the case when joined triple patterns contain bound subjects or objects, for example, $TP$’s
[?s p1 0] and [?s p2 ?b], where 0 is a bound object, the size of the joined triple pattern is a function of the upper bound size of the unbounded TP’s, namely [?s p1 ?a] and [?s p2 ?b] and the size of the bound object 0, which can be obtained from the data type specific histogram of p1 for 0.

In Hybridizer, we provide a user with two configurations for collecting statistics for joined triple patterns. The first configuration simply uses the number of triples belonging to predicates in the TP’s of a joined triple pattern to estimate the size of the joined pattern. While this configuration is less computationally intensive, since one only needs to collect the number of triples belonging to a predicate, it also leads to less accurate size estimations. The second configuration is designed to improve overall accuracy of size estimations, but it requires a significant amount of memory space and estimation time. The configuration maintains subject- and object-specific maps for each unique predicate, where the maps maintain a mapping between a unique prefix of a predicate and the number of triples containing that prefix. For example, given the triples t1, t2 and t3:

\[
\begin{align*}
t1: & \quad <http://example.org/pubs/A> \text{ dc:creator } <http://example.org/authors/A> \\
t2: & \quad <http://example.org/pubs/A> \text{ dc:creator } <http://example.org/authors/B> \\
t3: & \quad <http://example.org/authors/A> \text{ foaf:name } "ABC"\text{"xsd:string},
\end{align*}
\]

the subject map, sMap, and object map, oMap, for predicate dc:creator will contain the following entries:

\[
\begin{align*}
sMap: & \quad http://example.org/pubs/A \rightarrow 2 \\
oMap: & \quad http://example.org/authors/A \rightarrow 1 \\
& \quad http://example.org/authors/B \rightarrow 1,
\end{align*}
\]

while the subject map, sMap, for foaf:name will contain the following entry:

\[
\begin{align*}
sMap: & \quad http://example.org/authors/A \rightarrow 1.
\end{align*}
\]
Note that, there is no object map for foaf:name, since objects are literal values. Then, if one were to construct a joined triple pattern containing TP’s [?s dc:creator ?o] and [?o foaf:name ?n], we could use the prefix maps constructed above to accurately estimate the size of the joined pattern.

Statistics for other SPARQL operators: In Hybridizer, we support size estimation of BGP’s involving FILTER conditions of the form: FILTER (Var Op Value), where Var is a variable in the set V, Op is one of the operators: \{<,>,=,\le,\ge\}, and Value is a URI in the set I or a literal value in the set L. Since we constrain Op’s to specific operators and Value’s to URI’s or literals, we can use the previously defined data type specific histograms to efficiently estimate the size of FILTER conditions.

Since a BGP (which is a graph pattern) is composed of joined triple patterns and FILTER conditions, one can use the previously collected statistics to determine the size of a BGP. Furthermore, since the UNION and OPT operators are themselves defined over graph patterns, one could estimate the size of the resulting graph pattern using a recursive definition by employing the previously collected statistics.

A.1.2 Selectivity Estimation

In this subsection, we detail the framework used to perform RDF selectivity estimation. The framework is built on top of the statistics collection mechanisms for RDF data discussed in the previous subsection. As before, we first outline the selectivity estimation techniques for triple patterns and joined triple patterns. Then, we describe how selectivity estimation is performed for other SPARQL operators using the techniques defined for triple patterns/joined triple patterns. A summary of the selectivity estimation techniques for all RDF/SPARQL constructs is given in Table A.2.

Triple pattern selectivity: The selectivity of a triple pattern, TP, is estimated as 
\[ sel(TP) = sel(sp) \times sel(pp) \times sel(op) \], where \( sel(TP) \) denotes the selectivity for triple pattern
Table A.2. Summary of Selectivity Estimation for RDF

<table>
<thead>
<tr>
<th>RDF/SPARQL Constructs</th>
<th>Selectivity Estimation Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject Pattern, ( sp )</td>
<td>( \text{sel}(sp) = \frac{1}{R} ), ( R ) denotes the total resources</td>
</tr>
<tr>
<td>Predicate Pattern, ( pp )</td>
<td>( \text{sel}(pp) = \frac{T_{pp}}{T} ), ( T_{pp} ) denotes triples containing ( pp ), ( T ) denotes total triples</td>
</tr>
</tbody>
</table>
| Object Pattern, \( op \)  | \( \text{sel}(op) = \begin{cases} 
\text{sel}(p,op_c) & \text{if } p \text{ is bound} \\
\sum_{p_i \in P} \text{sel}(p_i,op_c) & \text{otherwise.} 
\end{cases} \)  

\((p,op_c)\) is the class of histogram for \( p \) in which \( op \) falls  
\( \text{sel}(p,op_c) = \frac{h_c(p,op_c)}{T_p} \), \( h_c(p,op_c) \) is the frequency of class \((p,op_c)\) |
| Triple Pattern, \( TP \)  | \( \text{sel}(TP) = \text{sel}(sp) \times \text{sel}(pp) \times \text{sel}(op) \)              |
| Joined Triple Pattern     | \( \text{sel}(P) = \frac{S_P}{T^2} \), \( P \) is the joined \( TP \) consisting of two \( TP \)'s |
| FILTER                    | \( \text{sel}(FP) = \text{sel}(C) \), where \( FP = (Var \ Op \ Value) \)                         
\( C = \{(p,op_c)|(p,op_c)\text{ is a histogram class for } p \text{ in which } op \text{ falls under } Op\} \)  
p represents the predicate in the \( TP \) containing \( Var \)  
\( Op \) represents the operator specified in \( FP \)  
\( op \) represents the object value, \( Value \), as specified in \( FP \)  
\( \text{sel}(C) = \frac{\sum_{(p,op_c) \in C} h_c(p,op_c)}{T_p} \)  
\( h_c(p,op_c) \) is the frequency of class \((p,op_c)\) |
| Basic Graph Pattern, \( BGP \)| All Previous                                                                                   |
| UNION                     | \( \text{sel}(gp_1) \cup \text{sel}(gp_2) \)                                                        |
| OPT                       | \( \text{sel}(gp_1) \)                                                                            |
TP, sel(sp) denotes the selectivity for the subject pattern sp, sel(pp) denotes the selectivity for the predicate pattern pp and sel(op) denotes the selectivity for the object pattern op. Note that, the selectivity estimate is a real value in the interval [0, 1] and denotes the size, i.e., the estimated number of triples that match a pattern, normalized by the total number of triples. As stated in (Stocker et al., 2008), the formulation above only approximates sel(TP) as it implicitly assumes that sel(sp), sel(pp) and sel(op) are statistically independent, which they will not be in most cases.

As before, we have two cases depending on whether sp, pp and op are bound or unbound. The case when they are unbound is trivial, as the selectivity in this case is 1.0, since an unbound sp, pp or op matches all triples in the RDF graph. On the other hand, when sp, pp or op is bound we require specific selectivity estimation techniques. The selectivity of a bound subject is estimated as \(sel(sp) = \frac{1}{R}\), where \(R\) denotes the total number of resources in the graph. The selectivity of a bound predicate is estimated as \(sel(pp) = \frac{T_{pp}}{T}\), where \(T_{pp}\) denotes the number of triples matching predicate pattern \(pp\) and \(T\) is the total number of triples in the graph. The selectivity of a bound object is estimated as:

\[
sel(op) = \begin{cases} 
sel(p, op_c) & \text{if } p \text{ is bound} \\
\sum_{p_i \in P} sel(p_i, op_c) & \text{otherwise}, 
\end{cases}
(A.1)
\]

where \((p, op_c)\) denotes the class of histogram for predicate \(p\) in which object pattern \(op\) lies, and \(sel(p, op_c) = \frac{h_c(p, op_c)}{T_p}\) denotes the number of triples matching \(op\) normalized by the number of triples that match \(p\). If \(p\) is unbound, the selectivity is estimated as a sum of \(sel(p_i, op_c)\) for every predicate \(p_i \in P\).

**Joined triple pattern selectivity:** The selectivity estimation for joined triple patterns uses statistics collected using one of the two configurations specified in the previous section. The collected statistics provide an upper bound size for the joined triple pattern consisting of triple patterns related with a common variable, having unbound subjects and objects.
Assuming that the upper bound size for a joined triple pattern $P$ is $S_P$, the selectivity of $P$ can be estimated as:

$$\text{sel}(P) = \frac{S_P}{T^2},$$  \hspace{1cm} (A.2)

where $T^2$ denotes the square of the total number of triples in the graph, and represents the size of the Cartesian product of the two triple patterns in the joined triple pattern, with unbound subjects and objects. The selectivity estimation for $P$ is corrected by a specific factor depending on whether the triple patterns in the joined triple pattern contain bound subjects or objects. The factor is a function of the selectivities of the bound subjects and objects in the triple patterns. For example, if $P$ consists of $TP$'s $[?s \ p_1 \ C]$ and $[?s \ p_2 \ ?o]$, the selectivity of $P$ is estimated as:

$$\text{sel}(P) = \frac{S_P}{T^2} \times \text{sel}(p_1, op_c),$$  \hspace{1cm} (A.3)

where $\text{sel}(p_1, op_c)$ is the selectivity of the bound object $C$ in the first triple pattern of $P$.

**Selectivity estimation for other SPARQL operators:** The selectivity of a FILTER condition, $FP = (Var \ Op \ Value)$, is estimated as follows: $\text{sel}(FP) = \text{sel}(C)$, where $C = \{(p, op_c)|(p, op_c) \text{ is a histogram class for } p \text{ in which } op \text{ falls under condition } Op\}$. Furthermore, $p$ denotes the predicate in the $TP$ containing the variable $Var$ in $FP$, $Op$ denotes the operator specified in $FP$ and $op$ represents the object value, $Value$, as specified in $FP$. For example, given a condition, $\text{FILTER}(?o \geq "25"^^xsd:integer)$ and the corresponding $TP$ $[?s \ swrc:pages \ ?o]$, $p$ is equal to $swrc:pages$, $Op$ is equal to $\geq$, and $op$ is equal to "$25"^^xsd:integer$. Furthermore, $C$ denotes the set of histogram classes for predicate $swrc:pages$ in which object "$25"^^xsd:integer$ lies under the $\geq$ condition, and $\text{sel}(C) = \frac{\sum_{(p, op_c) \in C} h_{x}(p, op_c)}{T_p}$ denotes the summation of frequencies of classes in $C$ normalized by the number of triples that match $p$.

Since a $BGP$ (which is a graph pattern) is composed of joined triple patterns and FILTER conditions, one can use the previously defined techniques to estimate the selectivity of a
BGP. Furthermore, since the UNION operator is defined over graph patterns, one could estimate the selectivity of the resulting graph pattern using a recursive definition as $sel(gp_1) \cup sel(gp_2)$. In the case of the OPT operator, graph pattern $gp_2$ does not contribute to the number of solutions in the result, and therefore, the selectivity of a graph pattern involving OPT can be estimated as $sel(gp_1)$.

A.2 Selectivity Estimation for Inference queries

In Hybridizer, we support selectivity estimation for inference queries containing class-subclass and property-subproperty relationships. To illustrate, consider the following ontology:

```latex
ex:p1 rdf:type owl:ObjectProperty;
ex:p2 rdf:type owl:ObjectProperty;
ex:p2 rdfs:subPropertyOf ex:p1
```

and the following inference query $Q$:

```sql
Q: SELECT ?s WHERE { ?s ex:p1 <http://ex.org/A> }.
```

To accurately estimate the selectivity of the $TP$ $[?s ex:p1 <http://ex.org/A>]$, one needs to consider the statistics for $ex:p2$, in addition to the statistics for $ex:p1$, since $ex:p2 rdfs:subPropertyOf ex:p1$. In Hybridizer, the selectivity of $TP$’s involving inference is estimated as a three step process. First, the subclasses/subproperties ($ex:p2$) associated with classes/properties ($ex:p1$) in $TP$ are identified using the supplied ontology. Second, the statistics of the discovered subclasses/subproperties ($ex:p2$) are combined with those of the original class/property ($ex:p1$) on-the-fly at runtime. Third, the selectivity of $TP$ is estimated using the updated statistics. Note that, estimating the selectivity of inference queries does not require any new statistics to be collected. Finally, although we have described the selectivity estimation process for inference queries on the basis of an example involving a single triple pattern, the process remains the same for more complex inference queries involving basic graph patterns and additional SPARQL operators.
APPENDIX B

JENA-MM: EFFICIENT RDF STREAM PROCESSING USING BUFFER MANAGEMENT ALGORITHMS

An RDF streaming application is used for real time processing of RDF data. The power of such an application lies in predicting new facts from existing data. To this end, an RDF streaming application must be able to support both, inference and non-inference queries. Furthermore, these applications do not know a priori the amount of data they will process. The data can be stored in memory so that query processing is fast, but this is not scalable. The data could be stored in a relational database allowing scalability, but leading to a longer query processing time. In this appendix, we present Jena-MM, a system that comprises two unified models to solve these problems. A unified inference model starts with in-memory storage and switches to disk based persistence using Lucene when the system begins to run out of memory. To determine which candidate nodes from a stream are to be left in memory, rather than using a random approach, Jena-MM uses a memory management algorithm to perform this task. On the other hand, a unified non-inference model further extends an inference model with disk based persistence using a relational database as the size of the stream gets even larger. A widely used open-source framework for building Semantic Web applications called Jena is extended with these unified models. This appendix is structured as follows: Section B.1 presents the motivation behind the development of Jena-MM, and the novel contributions in Jena-MM. Then, Section B.2 presents related work in the field highlighting the approaches that have been taken thus far based on in-memory and relational databases. In Section B.3 preliminary information is presented about the tools and techniques that are used in the unified models. Section B.4 discusses the proposed
unified models for solving issues of scalability and query response time. A comprehensive performance evaluation of Jena-MM is given in Section B.5.

B.1 Motivation

Semantic Web \cite{Berners-Lee2001} allows a user access to information as well as to the metadata describing this information. The metadata facilitates automated processing of information by computers and can be represented in the same format as the information using the Resource Description Framework (RDF). RDF represents data as a triple composed of a subject, predicate and object, the semantics of this being, “the subject takes the value of object for the predicate”.

Streaming data has become an integral part of the World Wide Web (WWW) with real-time data sources such as Facebook\footnote{http://en.wikipedia.org/wiki/World_Wide_Web} and Twitter\footnote{http://www.facebook.com/}\footnote{http://twitter.com/}. There are several reasons as to why these applications would benefit from being integrated with each other as well as with existing Semantic Web data. Such an integration could lead to the ability to ask continuous queries that can be answered in real time. Further, data from several real-time sources could be combined in novel ways to build personalized services. There are several differences between regular RDF data and RDF streams. Regular RDF data is created and materialized once and there are only a few sporadic changes to this data over time. On the other hand, RDF streaming data is continuously created like sensor data. This streaming RDF data must be continually materialized and should be available for querying immediately. Some additional motivations for the use of RDF streams are the ability to ask continuous queries, a smaller memory footprint at query time, selective indexing of data and pipelined processing.
An example of a real-time RDF streaming application could be based on an existing application such as Twitter. For example, consider one such dummy application, called MyTwitterRDF, described here. A representation of MyTwitterRDF in the Semantic Web includes a terminology box (TBox) consisting of a set of classes and properties that model online social networks. A combination of several existing vocabularies and ontologies such as FOAF, SIOC, RELATIONSHIP, DC, SKOS etc. can be used to construct this TBox. An assertion box (ABox) for MyTwitterRDF containing the streaming RDF data can then be continually created based on the TBox and the real-time data available through the Twitter API. The TBox introduces the terminology, i.e., the vocabulary of an application domain, while the ABox contains assertions about named individuals in terms of this vocabulary (Baader and Nutt, 2003). Privacy policies for users in MyTwitterRDF can be modeled using SWRL (Horrocks et al., 2004) rules in the TBox. Queries to MyTwitterRDF for finding who has access to user information (an inference query) require a combination of the ABox with the specified rules using inference before they are answered. On the other hand, queries to MyTwitterRDF for finding user profile information (a non-inference query) can be directly answered using the ABox that has been materialized upto the point at which the query was asked. The main requirement of MyTwitterRDF would be to store the theoretically unlimited amount of RDF data while being able to answer queries as soon as the data is materialized. A general solution to this requirement would be to store the RDF data in a memory backed triple store in which case answering queries is a simple task, but only a

4 http://xmlns.com/foaf/spec/
5 http://rdfs.org/sioc/spec/
6 http://vocab.org/relationship/.html
8 http://www.w3.org/TR/2008/WD-skos-reference-20080829/skos.html
9 https://dev.twitter.com/
limited amount of data can be stored. Another approach is to store the RDF data in one
of the available disk based persistence stores giving scalability, but at the cost of a reduced
query time.

The above application is an example of a more general problem of query execution speed 

versus storage capability. This problem is particularly difficult when we are bounded by
real time requirements but do not know a priori the size of the data that will be processed.
Existing RDF toolkits such as Sesame (Broekstra et al., 2002) and Jena (McBride, 2002)
allow efficient creation and manipulation of RDF graphs using one of in-memory or disk 

backed persistence. However, the requirements of an RDF streaming application do not fit 

completely in any of these approaches.

The main goal of Jena-MM (Khadilkar et al., 2010) is to present a unified model to solve this problem by combining in-memory storage with disk based persistence. Such a unified model can be used to handle the storage and query constraints of an RDF streaming application such as MyTwitterRDF. In this appendix, two separate unified models are presented, one of which is optimized to handle inference queries while the other is optimized to manage non-inference queries. Both unified models are implemented as Jena graphs that combine in-memory and disk based storage. The inference model combines Jena’s in-memory model with a Lucene (Hatcher et al., 2008) model, while a non-inference model combines Jena’s in-memory model with a Lucene model and Jena’s database model. Lucene is an open-source tool written in Java that can be used for text indexing and searching. Lucene’s high performance and ease of integration with any application are some of the motivating factors for its use. The design goals of the unified model with respect to RDF streams can be summarized as follows:

10https://github.com/vaibhavkhadilkar/jena-mm
11We acknowledge that some of the preliminary details about Jena-MM were presented in (Khadilkar et al.,
2010), which is available at: http://ceur-ws.org/Vol-658/
• The unified model supports real-time queries, \textit{i.e.,} queries can be asked as soon as data is materialized.

• The size of the RDF stream that can be handled by the unified model depends on the type of the model (inference or non-inference). The inference model can theoretically store an unlimited RDF stream while providing query performance that is better than Jena’s database models. The non-inference model is best suited for intermediate sized RDF streams (tens of millions of triples) beyond which Jena’s database model provides a better query performance.

The unification of Jena’s in-memory model with the Lucene model is discussed first since it is common to both unified models. The unified models initially add triples from the RDF stream to in-memory storage and Jena-MM switches to a Lucene model when it begins to run out of memory. A Lucene model’s storage schema is a replica of Jena’s in-memory model; three Lucene indices are created, each of which is indexed by \textit{subjects}, \textit{predicates} and \textit{objects}. The system uses one of the following Lucene model creation strategies to optimize query performance for inference and non-inference queries respectively: (1) All indices are created at the same time. (2) Each index is created individually as needed.

It should be evident that simply taking the RDF data from in-memory storage and persisting it to a Lucene model will not work. This is because some nodes from the stream will be frequently accessed while others will be central actors in the stream. To determine which nodes from the stream to leave in memory, Jena-MM uses different buffer management algorithms. Memory management algorithms such as first-in-first-out (FIFO), last-in-first-out (LIFO) \textit{etc.} are used to keep relevant nodes in memory based on the frequency of their access patterns. To keep nodes in memory based on their centrality in the stream, the system uses degree centrality (DC) and individual clustering coefficient (CC). The inference model only uses this combination of in-memory and Lucene models. For a non-inference model,
Jena-MM switches from a Lucene model to Jena’s database model when a threshold limit is reached. This is done because beyond the threshold limit the database model provides optimal performance.

In this work, the unified models are only compared with Jena (McBride 2002; Wilkinson et al. 2003b) since the system is designed within the Jena framework. Further, other database approaches such as Sesame (Broekstra et al. 2002) are not explored because the primary objective of this work is to create a unified model for storing and querying RDF streams rather than comparing different existing tools.

B.1.1 Our Contributions

To address the general challenge of query execution speed versus storage capability in RDF streams, Jena-MM presents the following novel contributions:

• A unified inference model that adapts existing in-memory/disk-based solutions to store a theoretically unlimited RDF stream while providing excellent query response times.

• A unified non-inference model that combines in-memory and disk based (Lucene and relational database) persistence that is ideal for intermediate sized RDF streams.

• Effective use of traditional buffer management algorithms and social network measures to select “relevant” nodes in a graph to be retained in memory.

• The development of an open-source software tool\textsuperscript{12, 13} that combines all algorithms and Lucene model construction strategies used with the unified models.

\textsuperscript{12}http://jena.sourceforge.net/contrib/contributions.html

\textsuperscript{13}http://cs.utdallas.edu/semanticweb
B.2 Literature Review

Jena-MM intends to find an efficient way to store and query RDF streams by combining various RDF storage and buffer management techniques.

To date, most of the work related to RDF streams has concentrated on querying streams effectively (Bolles et al., 2008; Groppe et al., 2007). To the best of our knowledge, this is the first work focused on storing as well as querying RDF streams effectively. Currently, RDF streams can only be stored using the persistence mechanism of tools such as Sesame (Broekstra et al., 2002), AllegroGraph (Franz-Inc., 2005) and Jena (Wilkinson et al., 2003b) due to the scalability offered by them. These tools mainly use relational databases as the persistence mechanism due to their numerous advantages such as scalability, fault tolerance, etc. Excellent surveys on storing RDF data in relational databases have been given in (Beckett, 2002; Beckett and Grant, 2003). To date, there are three main approaches for storing RDF data in relational databases. The first approach uses a generic representation scheme, in which a triple table is created to store an RDF stream (Broekstra et al., 2002; Guha, 2001; Beckett, 2001). The second approach uses a more specific representation scheme, in which, in addition to the triple table, several other tables are created to efficiently store and query triples (Wilkinson et al., 2003b; Alexaki et al., 2001; Pan and Heflin, 2003). A third approach is the vertically partitioned scheme in which the triple table is partitioned into $m$ two column tables where $m$ unique predicates exist in the RDF graph (Abadi et al., 2007). Some disadvantages of these schemes are, a longer processing time for complex queries requiring multiple joins, the difficulty of updating data since multiple tables and indices may need to be updated, and a longer query response time for a small RDF stream.

Research has also focused on non-database approaches to store RDF data. An approach based on a “graph database” is found in AllegroGraph RDFStore (Franz-Inc., 2005) where the triple table can be filled with any subject, predicate, object, and graph values, thus allowing the representation of any graph structure. BigOWLIM (Ontotext-AD, 2004) stores a part
of the RDF graph in memory, while the rest is maintained in N-Triple files on disk. Another approach is to use memory to store the RDF stream like Jena; however, since memory is expensive, the size of the stream that can be stored is small. Therefore, in Jena-MM, we propose a unified approach, which extends the in-memory storage of Jena with temporary disk based persistence using Lucene, and also uses relational databases if necessary. As suggested before, we want to keep relevant nodes in memory at all times. To perform this task, we investigated various memory management techniques available in the literature.

A lot of research has been devoted to the study of memory management algorithms. Excellent surveys for memory management techniques in relational databases can be found in (Chou and DeWitt 1985; Effelsberg and Härder 1984). Three different approaches have been taken for memory management in relational databases. The first approach allocates memory at runtime; virtual memory algorithms (Mattson et al. 1970) and domain separation algorithms (Reiter 1976; Denning 1968) are examples of this approach. The second approach uses prior knowledge of the kinds of queries that will be posed to allocate memory. Examples of this approach are the hot set algorithm (Sacco and Schkolnick 1982) and the DBMIN algorithm based on the query locality set model (Chou and DeWitt 1985). The third approach is a hybrid of the previous two approaches. An algorithm based on this approach is the flexible buffer allocation based on marginal gains (Ng et al. 1991). Since we want to pick the vital nodes in an RDF stream, we have also examined some simple social network centrality measures such as degree centrality (DC) and individual clustering coefficient (CC) (Jackson 2008) to perform this task. We use these algorithms to determine the nodes to be left in memory, while the remaining nodes are persisted to disk.

B.3 Background Information

In this section, we present a brief description of specific tools and technologies used in Jena-MM, which were not presented in Chapter 2. In particular, a brief overview of Lucene is
presented, which is followed by a short introduction to centrality metrics from social network analysis that have been adapted into memory management algorithms in Jena-MM.

**B.3.1 Overview of Lucene**

Lucene\(^1\) is a text indexing and searching API. A single record in the index is called a *Document*, which contains a list of fields; each field has a name and a value. A *Term* is the smallest unit of indexing, which is usually a word of text. An *Analyzer* is used to prepare text that needs to be indexed; several analyzers are available with users having the option of plugging in their own analyzers. Additionally, Lucene provides word stemming during indexing with one of several available stemmers. Lucene also provides search capabilities through the *IndexReader* and *IndexSearcher* classes. An *IndexReader* provides an interface for accessing an index while an *IndexSearcher* implements search capabilities over an *IndexReader*. Filtering is also available through *TermEnum* (enumerate all *Terms* in the index for a given field) and *TermDocs* (enumerate all *Documents* that contain a given *Term*).

**B.3.2 Overview of Centrality Measures**

Centrality measures provide a good way of estimating central actors in a social network. Since central nodes from an RDF stream could be vital to queries in the unified models, these nodes should be kept in memory for as long as possible. A brief summary is now provided of two centrality measures that have been used in Jena-MM, namely degree centrality and individual clustering coefficient.

Degree centrality is the simplest measure of the position of a node in a graph (Jackson, 2008). In a graph with \( n \) nodes, if a node has degree \( n - 1 \), it is connected to all other nodes and hence is very central. Degree centrality for a node \( i \), \( DC_i \), is defined as follows:

\[
DC_i = \frac{d_i(g)}{n - 1}
\]

\(^1\)http://lucene.apache.org/java/docs/index.html
where $d_i(g)$ denotes the degree of $i$ in $g$.

The individual clustering coefficient is a measure of the connectivity of a node’s neighbors (Jackson [2008]). If all of a node’s neighbors are connected then that node is central in its neighborhood. The individual clustering coefficient for a node $i$ in a graph $g$, $CC_i(g)$, is calculated as follows:

$$CC_i(g) = \frac{\# \{jk \in g | k \neq j, j \in N_i(g), k \in N_i(g) \}}{d_i(g)(d_i(g) - 1)/2}$$ (B.2)

where $CC_i(g)$ denotes the individual clustering coefficient of $i$ in $g$, $N_i(g)$ denotes the nodes in the neighborhood of $i$ in $g$ and $d_i(g)$ denotes the degree of $i$ in $g$.

B.4 Jena-MM Architecture

In this section, we present an architectural overview of Jena-MM, which comprises a unified Jena model that stores and queries RDF streams. For a better understanding, a combined view of creating a unified model in the non-inference and inference scenarios is given while different views are shown for an inference and non-inference query.

B.4.1 Jena-MM Storage Architecture

Figure [B.1] shows a flow of control for storing RDF streams in Jena-MM. We first check if an application wants to create an inference or non-inference model (step (1)). For the inference case, a unified model consists of in-memory and Lucene models, while for the non-inference case a unified model comprises in-memory, Lucene and database models. The in-memory and Lucene components are presented first since they are common to both models. This is followed by a discussion of the architectural components that differ for both models.

**In-memory and Lucene models:** For both Jena-MM unified models, the system starts out by adding triples from an RDF stream to an in-memory model (step (3b)). When a triple is added to an in-memory model, statistics for it are maintained in an in-memory buffer. The
simplest way to create a buffer is based on storing the following statistics for every unique subject in a stream: (1) The subject’s degree. (2) A timestamp of when the subject was last accessed. (3) A pointer to the triples belonging to the subject. Note that, the statistics are updated for every subject of each incoming triple that is added to an in-memory model. A buffer could also be constructed in more complicated ways using various combinations of subjects, predicates and objects. These other approaches are not explored since for example, a predicate would be artificially central in an RDF stream since it has a very large degree. Similarly, an object would not be central in a stream since it has a very small degree. When the number of triples added to an in-memory model reaches the writeThreshold (step (4)), the buffer is input to the buffer management subsystem. The writeThreshold is given by the following equation,

\[
writeThreshold = initThreshold \times totalMem,
\]
where \textit{initThreshold} and \textit{totalMem} are user-defined input parameters. The \textit{initThreshold} is an initial estimate of the number of triples that can be added to an in-memory model per GB of memory without causing memory overflow exceptions. For example, a value of 100,000 for \textit{initThreshold} denotes that a user estimates that 100,000 triples can be added to an in-memory model per GB of available memory. The \textit{totalMem} represents the integer value of the total memory (given in GB) that is available to the current invocation of the unified model. As an example, if the unified model is invoked in an application with 5GB main memory, then \textit{totalMem} = 5. Clearly both these parameters can be supplied by a user based on their knowledge of the application environment.

The buffer management subsystem then sorts a buffer based on a selected memory management algorithm (step (5)). A selected algorithm uses the \textit{degree} and \textit{timestamp} values to sort the buffer. Several traditional memory management algorithms such as first-in-first-out (FIFO), last-in-first-out (LIFO), least-recently-used (LRU), most-recently-used (MRU) and RANDOM have been tested in Jena-MM. Additionally, traditional social network analysis centrality metrics such as degree centrality (DC) and individual clustering coefficient (CC) have also been adapted as memory management algorithms. The unified models then use the sorted buffer in conjunction with a selected Lucene model construction strategy to write triples to a Lucene model by following the \textit{pointers} maintained by a buffer (step (6)).

Figure B.2 shows the storage schema used in the creation of a Lucene model. A Lucene model comprises three separate Lucene indices, one each for \textit{subjects}, \textit{predicates} and \textit{objects}. The idea of using three indices is inspired by Jena’s in-memory model that maintains three separate data structures, one each for \textit{subjects}, \textit{predicates} and \textit{objects}. Each Lucene index consists of a set of \textit{Documents} represented in Figure B.2 by the variables $d_1, d_2, \ldots, d_n$. Note that, these variables are only used in Figure B.2 for clarity; \textit{Documents} do not contain any explicit identifier when they are added to a Lucene index. Every \textit{Document} in a Lucene model consists of the following three fields: (1) \textit{id} – An explicit identifier for a \textit{Document}
that is used when Jena-MM wants to retrieve all Documents in an index. (2) uri – The URI of the node being indexed by the current Document. (3) triples – A list of triples that belong to the node being indexed by the current Document.

Two different Lucene model construction strategies have been tested in Jena-MM: The first creates all Lucene indices at the same time (C-S: cache-subject), while the second creates Lucene indices one by one, as needed (C-S-Eff: cache-subject-efficient). The process of writing triples to a Lucene model continues until a user defined fraction, memoryFactor, of writeThreshold is left in memory. Note that, memoryFactor is entered as an integer between 1 and 100, so that the number of triples left in memory at the end of each invocation of the method that writes triples to a Lucene model is memoryFactor% of writeThreshold.

A description of the architectural components that are specific to the inference and non-inference models is now given.

**Inference model:** For an inference model, Jena-MM first checks if the current triple being read from a stream is part of a TBox (step (2a)). If this requirement is satisfied, then the current TBox triple is stored in memory (step (3a)). On the other hand, if the triple is an ABox triple, we store it in memory and cache the statistics for that triple in an in-memory buffer (step (3b)). Note that, TBox triples are always kept in memory since the number of TBox triples is much lesser than ABox triples (Hogan et al., 2009). Jena-MM distinguishes between TBox and ABox triples and ensures that TBox triples always remain in memory.
using the following procedure: Jena-MM first reads all TBox triples into an inference model. While these triples are being read, a flag ensures that statistics for them are not maintained in the in-memory buffer. Only nodes whose statistics are maintained in the buffer can be persisted to a Lucene model based on the selected memory management algorithm. In this way, the flag ensures that TBox nodes are not persisted to a Lucene model.

**Non-inference model**: For a non-inference model, the system begins by checking if \( dbThreshold \) is reached (step (2b)). \( dbThreshold \) signifies a saturation point for a Lucene model and the unit for \( dbThreshold \) is number of triples. \( dbThreshold \) is computed as a function of the size of the RDF stream read so far (\# of triples), and the two input parameters, \( initThreshold \) and \( totalMem \), defined above. Additional details of the computation of \( dbThreshold \) will be made available in Section B.5. If \( dbThreshold \) is reached, Jena-MM checks if a database model has been previously created (step (3a)). If it has, the current triple is stored in that model (step (4a')). If it has not been created, it is first created and then, the current triple is stored in the newly created database model (step (4b')). Additionally, a separate thread is started to copy triples from the in-memory and Lucene models to the newly created database model in the background. Note that, once a database model is created, all incoming triples from the current RDF stream will be stored in this model. On the other hand, if \( dbThreshold \) is not reached, the triple is stored in memory and its statistics are cached in an in-memory buffer (step (3b)).

Table B.1 provides a summary of parameters used in Jena-MM.

### B.4.2 Query Processing in Jena-MM

Figure B.3 shows the process of querying the Jena-MM unified models with an inference (Figure B.3(a)) or non-inference query (Figure B.3(b)). An inference/non-inference query involves searching a stream for arbitrarily complex triple patterns; the difference between them being that an inference query requires an ontology in addition to data for answering
Table B.1. Variables used in Jena-MM

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbThreshold</td>
<td>Threshold from where the database model is used</td>
</tr>
<tr>
<td>initThreshold</td>
<td>Initial estimate of number of triples that can be added to in-memory model per GB of main memory</td>
</tr>
<tr>
<td>totalMem</td>
<td>Integer value of total memory (in GB) available to the current run</td>
</tr>
<tr>
<td>writeThreshold</td>
<td>Threshold value after which a fraction of the triples are written to the Lucene model</td>
</tr>
<tr>
<td>memoryFactor</td>
<td>The fraction of writeThreshold (in terms of number of triples) that is left in memory</td>
</tr>
</tbody>
</table>

the query. Note that, when a query is issued to the unified models, Jena-MM answers that query only over the currently stored stream.

For an inference query, a reasoner (Jena-MM currently uses the Pellet reasoner \cite{Sirin2007}) infers additional triples by reasoning over ABox triples using TBox triples. TBox triples are always in memory (steps (2) and (3)), while ABox triples are obtained by querying the in-memory and Lucene models (steps (4),(4a) and (5)). The reasoner then uses the original and inferred triples to return the result to a user (steps (6) and (7)). Note that, Jena-MM needs to query both, the in-memory and Lucene models for any given query, since triples associated with a node could be split between these models. Consider the following scenario: At a certain time (say $t_1$), triples associated with a node (say $n_1$) could have been written to a Lucene model. Then, at a later time (say $t_2$ such that $t_2 > t_1$), new triples for the same node ($n_1$) could be received from the RDF stream. These new triples are inserted into the in-memory model. Now, if a query is issued at this moment to return all triples belonging to $n_1$, a complete result can only be obtained when both, the in-memory and Lucene models are queried.

For a non-inference query, Jena-MM begins by checking if a database model has been created. If this condition is satisfied, then the query is simply passed to the database model
Figure B.3. Query Processing in Jena-MM
and the result obtained from that model is returned to a user (steps (2), (3) and (4)). If the database model has not been created, the query is passed to both, the in-memory and Lucene models (steps (2a) and (3a)). The combined result obtained from both models is then returned to a user application (step (4)).

B.5 Performance Evaluation

This section presents an experimental evaluation of Jena-MM using two SPARQL performance benchmarks. A brief description of the experimental setup and benchmarks is first presented. This is followed by a study of the following two measures: (1) The impact of various memory management techniques on query performance. (2) The effect of different Lucene model construction strategies on query performance.

B.5.1 Experimental setup

All experiments were conducted using Linux Ubuntu v8.04 Hardy Heron, on an Intel Core2 Duo E8400 3GHz CPU with 3GB DDR2 667 MHz memory and a 250GB Western Digital WDC WD2500AAJS-7 SATA hard drive with a 6MB cache. The experiments used JRE v1.6.0_03 as the Java engine while MySQL v5.1.4.1-3ubuntu12.6 was used for the database models. The experiments also used Lucene v2.3.1 to implement Lucene models.

B.5.2 Experimental Benchmarks

Since the unified models handle both, inference and non-inference queries, the experiments used the LUBM (Guo et al., 2005) and SP²Bench (Schmidt et al., 2009) benchmarks. LUBM and SP²Bench are SPARQL language benchmarks, LUBM is specifically used to test inference while SP²Bench is used to check non-inference query execution. LUBM consists of a

\[\text{http://www.mysql.com/}\]
realistic university ontology, a data generator for generating OWL data sets that can be scaled to arbitrary sizes, a set of benchmark queries and several performance metrics. On the other hand, SP²Bench is set in the DBLP (Ley and Reuther, 2006) library scenario and consists of both, a data generator for creating arbitrarily large DBLP-like documents, and a set of benchmark queries.

B.5.3 Experimental Procedure

The experiments used two separate threads to simulate storage and retrieval of an RDF stream. The first thread was used to load a dataset into a unified model (non-inference or inference), while the second thread was used to query the dataset at intervals of 5 seconds. The graph size used for SP²Bench was 50168 triples while for LUBM it was 1 university (∼ 103000 triples). Further, the parameters used in the unified models were set as follows: (1) initThreshold = 3/4 × the number of triples in the graph. (2) totalMem = 1, which implies that each run was given a memory limit of 1GB by restricting the JVM. (3) memoryFactor = no. of triples in mem/10, which means that 10% of the total triples were always left in memory. Finally, all experiments were conducted using only those benchmark queries for which the running times were less than 5 minutes using Jena’s in-memory model. As a result, Q4 from SP²Bench and Q2, Q7, Q8 and Q9 from LUBM were not used in any of the experiments. A timeout of 60 minutes was also set for every query used in the experiments. Note that, all parameter configurations and query selections described so far are applicable to experiments conducted in Sections B.5.4 and B.5.5. The parameter configurations and queries used to test the scalability of the unified models are presented in Section B.5.6.

B.5.4 Comparison of memory management algorithms

In this subsection, we present a performance comparison of various memory management algorithms using the SP²Bench and LUBM benchmarks. The goal of these experiments was to find the best memory management algorithm from amongst the available algorithms.
The choice of memory management algorithm used in the buffer management subsystem plays a significant role in query execution performance. Using these algorithms, if the system can leave nodes in memory that are frequently accessed and/or are central in the RDF stream, a user will always have excellent query times. Traditional virtual memory algorithms such as FIFO, LIFO, etc. \cite{Mattson1970} are used in relational databases to keep relevant pages in memory. These algorithms were adapted in the unified models to keep relevant nodes in memory based on the frequency of access patterns of all nodes in an RDF stream. Centrality measures such as degree centrality (DC) and individual clustering coefficient (CC) are traditionally used to determine central actors in a social network \cite{Jackson2008}. These algorithms were adapted in the unified models to find “important” nodes in an RDF stream, which are subsequently kept in memory.

Jena-MM uses timestamp values of all nodes maintained by the buffer to implement memory management algorithms. As an example, if the LRU algorithm is to be used, the buffer is sorted in increasing order of timestamp values. Then, Jena-MM selects one node at a time from the top of the buffer, and persists all triples belonging to that node to a Lucene model. This procedure is repeated until a user-defined fraction (memoryFactor) of the total triples remain in memory. Other virtual memory algorithms use timestamp values as variations of this technique.

Degree centrality (DC) is a simple measure of a node’s position in a graph \cite{Jackson2008}. Since DC is usually normalized between 0 and 1, if a node $i$ has $DC_i \rightarrow 1$, then it is ‘central’ in the stream and hence is a good candidate to be left in memory, while a node $i$ with $DC_i \rightarrow 0$ is a good candidate to be written to a Lucene model. The individual clustering coefficient (CC) is a measure of how well-connected a node is with its neighbors \cite{Jackson2008}. Because CC ranges between 0 and 1, in the context of the unified models, if a node $i$ has $CC_i \rightarrow 1$, then that node $i$ is considered to be central in the RDF stream and is left in memory. A node $i$ with $CC_i \rightarrow 0$ is a candidate node to be written to a Lucene
model. Jena-MM uses the degree values of all nodes maintained by the buffer to implement both these algorithms.

Further, the traditional memory management algorithms, namely LRU and MRU, were also combined with both, DC and CC in the following way. The timestamp values were first used to sort the buffer and if there was a tie where nodes were accessed in the same second, DC or CC were used to break the tie using the degree values of all tied nodes. The reader should note that DC and CC cannot be combined with either FIFO or LIFO since these algorithms produce a strict ordering of nodes based on timestamp values.

**Comparison of memory management algorithms for SP²Bench**

In this section, we compare various memory management algorithms using the SP²Bench benchmark. Since LRU and MRU were also combined with DC and CC, we first compare these algorithms to determine the best LRU/MRU-based algorithm. This is followed by a comparison of the best LRU/MRU-based algorithm with the remaining algorithms to find the best overall algorithm.

**Comparison of LRU/MRU-based algorithms for SP²Bench**

_Aim:_ The goal of this set of experiments was to determine the best LRU/MRU-based algorithm using the SP²Bench benchmark.

_Procedure:_ In these experiments, we measured the time required to execute different queries of the SP²Bench benchmark (queries other than Q4) on a graph size of 250137 triples for various LRU/MRU-based algorithms.

_Observations:_ Figures B.4–B.7 show a comparison of LRU/MRU-based algorithms for SP²Bench. We observe that LRU/MRU with degree centrality (LRU+DC/MRU+DC) perform better than plain LRU/MRU and LRU/MRU with individual clustering coefficient (LRU+CC/MRU+CC). This is due to the effectiveness of degree centrality in keeping central nodes of the SP²Bench dataset in memory. Jena-MM can directly access these central
Figure B.4. Comparison of LRU-based algorithms for SP²Bench: Queries Q1–Q6
Figure B.5. Comparison of LRU-based algorithms for SP²Bench: Queries Q7–Q12
Figure B.6. Comparison of MRU-based algorithms for SP²Bench: Queries Q1–Q6
Figure B.7. Comparison of MRU-based algorithms for SP²Bench: Queries Q7–Q12
nodes from memory without having to repeatedly query a Lucene model. This allows a faster processing of SP²Bench queries using LRU/MRU with degree centrality than the other variations of LRU/MRU.

**Comparison of all algorithms for SP²Bench**

**Aim**: The goal of this set of experiments was to determine the best overall algorithm using the SP²Bench benchmark.

**Procedure**: In these experiments, we measured the time required to execute different queries of the SP²Bench benchmark (queries other than Q4) on a graph size of 250137 triples for all memory management algorithms.

**Observations**: Figures B.8 and B.9 show a comparison of all memory management algorithms for SP²Bench queries. We observe that degree centrality (DC) produces the best performance from amongst all memory management algorithms. This is because DC keeps nodes that are relevant to SP²Bench queries in memory. Since these “central” nodes are available in memory, a non-inference model can directly use them during query processing. Additionally, Jena-MM does not need to process any triples from a Lucene model for these nodes. Due to these two reasons, a non-inference model that uses degree centrality (DC) as its memory management algorithm produces an optimal performance for non-inference queries.

**Comparison of memory management algorithms for LUBM**

In this section, we compare various memory management algorithms using the LUBM benchmark. Since LRU and MRU were also combined with DC and CC, we first compare these algorithms to determine the best LRU/MRU-based algorithm. This is followed by a comparison of the best LRU/MRU-based algorithm with the remaining algorithms to find the best overall algorithm.

**Comparison of LRU/MRU-based algorithms for LUBM**

**Aim**: The goal of this set of experiments was to determine the best LRU/MRU-based algorithm using the LUBM benchmark.
Figure B.8. Comparison of all algorithms for SP²Bench: Queries Q1–Q6
Figure B.9. Comparison of all algorithms for SP²Bench: Queries Q7–Q12
Procedure: In these experiments, we measured the time required to execute different queries of the LUBM benchmark (queries other than Q2, Q7, Q8 and Q9) on a graph size of \(\approx 103000\) triples for various LRU/MRU-based algorithms.

Observations: Figures B.10 and B.11 show a comparison of LRU/MRU-based algorithms for LUBM. We observe that plain LRU/MRU perform better than LRU/MRU with degree centrality (LRU+DC/MRU+DC) and LRU/MRU with individual clustering coefficient (LRU+CC/MRU+CC). The centrality based variants of LRU and MRU, namely LRU+DC/MRU+DC, LRU+CC/MRU+CC, use degree centrality or individual clustering coefficient to break ties between nodes that were created/accessed in the same second. Therefore, these algorithms leave nodes with a better centrality in memory as compared with LRU/MRU. However, these central nodes are not important to LUBM queries. The LUBM queries actually prefer nodes left in memory based on frequency of access patterns by the LRU/MRU algorithms, which in turn leads to a better performance for these algorithms.

Comparison of all algorithms for LUBM

Aim: The goal of this set of experiments was to determine the best overall algorithm using the LUBM benchmark.

Procedure: In these experiments, we measured the time required to execute different queries of the LUBM benchmark (queries other than Q2, Q7, Q8 and Q9) on a graph size of \(\approx 103000\) triples for all memory management algorithms.

Observations: Figure B.12 shows a comparison of all memory management algorithms for LUBM queries. We observe that plain MRU performs the best amongst all memory management algorithms. This is due to the fact that MRU moves the most recently used nodes along with their associated triples to a Lucene model. The “older” nodes that remain in memory are more important to LUBM queries. Further, these “older” nodes can be directly used by the inference API during the process of inference since they are always in memory. Also, the inference API does not need to process triples belonging to “older” nodes from a Lucene
Figure B.10. Comparison of LRU-based algorithms for LUBM: Queries Q1–Q14
Figure B.11. Comparison of MRU-based algorithms for LUBM: Queries Q1–Q14
Figure B.12. Comparison of all algorithms for LUBM: Queries Q1–Q14
model. Due to these reasons, an inference model with the MRU algorithm produces the best performance from amongst all memory management algorithms for inference queries.

B.5.5 Comparison of Lucene model construction strategies

The set of experiments conducted in the previous subsection used the default method of constructing a Lucene model. In this method, all three indices that make up a Lucene model are created at the same time (C-S: cache-subject). Note that, each index is built for subjects, predicates and objects, and it holds a set of Documents that contains the following three fields: (1) A unique identifier for every Document. (2) The URI of the node being indexed. (3) A list of triples that belong to the node being indexed.

As seen from Figures B.13–B.15 (the second column from the left), the default Lucene model creation technique produces mixed results. Note that, in Figures B.13–B.15, the memory management algorithms used for selecting nodes to be retained in memory are the previously determined best algorithms, namely DC for SP²Bench and MRU for LUBM. For SP²Bench queries, the default approach is very slow compared to Jena’s in-memory and database backed models, however, it is better than purely writing all triples to a Lucene model. On the other hand, for LUBM queries, the same approach is comparable to Jena’s in-memory model and it outperforms the database backed models and the purely Lucene model approach. The other approaches are slower since Jena performs inference in memory during which multiple queries need to be made to the underlying database or Lucene models for ABox triples.

In order to rectify the situation for SP²Bench queries, the following approach was taken based on domain separation algorithms (Reiter, 1976; Denning, 1968). The buffer is built as before but instead of creating all Lucene indices at the same time, they are created one at a time as needed (C-S-Eff: cache-subject-efficient). The set of equations (B.4, B.5 and B.6) given below help Jena-MM to determine when each index is to be created.
Figure B.13. Comparison of persistence strategies for SP²Bench: Queries Q1–Q6 (M denotes an Out Of Memory exception)
Figure B.14. Comparison of persistence strategies for SP²Bench: Queries Q7–Q12 (M denotes an Out Of Memory exception)
Figure B.15. Comparison of persistence strategies for LUBM: Queries Q1–Q14 (T denotes a Timeout exception)
\[ s + p + o \geq a \times \text{writeThreshold} \rightarrow \text{writeP} \]
\[ 3p \geq a \times \text{writeThreshold} \rightarrow \text{writeP} \hspace{1cm} (\because s = p = o) \]
\[ p \geq (a/3) \times \text{writeThreshold} \rightarrow \text{writeP} \]
\[ p \geq (a/3) \times \text{initThreshold} \times \text{totalMem} \rightarrow \text{writeP} \]
\[ \frac{\text{graph size}}{(a/3) \times \text{initThreshold}} \geq \text{totalMem} \rightarrow \text{writeP} \hspace{1cm} (\because p = \text{graph size}) \hspace{1cm} (B.4) \]

\[ s, p \text{ and } o \text{ denote the size of the in-memory subject, predicate and object structures, in triples respectively, and } \text{writeP} \text{ denotes the method to build the predicate index. When the point of creating the predicate index is reached, } s = p = o, \text{ because each triple is indexed by } s, p \text{ and } o. \text{ Also, } p = \text{graph size}, \text{ since } p \text{ represents the number of triples that have been added to the graph. Similarly to create the object index,} \]

\[ s + o \geq b \times \text{writeThreshold} \rightarrow \text{writePO} \]
\[ 2o \geq b \times \text{writeThreshold} \rightarrow \text{writePO} \hspace{1cm} (\because s = o) \]
\[ o \geq (b/2) \times \text{writeThreshold} \rightarrow \text{writePO} \]
\[ o \geq (b/2) \times \text{initThreshold} \times \text{totalMem} \rightarrow \text{writePO} \]
\[ \frac{\text{graph size}}{(b/2) \times \text{initThreshold}} \geq \text{totalMem} \rightarrow \text{writePO} \hspace{1cm} (\because o = \text{graph size}) \hspace{1cm} (B.5) \]

where \text{writePO} \text{ is the method used to create and write triples to the predicate and object indices. At the point of creating the object index, } s = o, \text{ based on an explanation similar to the one given after Equation B.4. Also, } (o = \text{graph size}) > p \text{ because } o \text{ signifies the number of triples added to the graph and the current equation is used after the predicate index has already been created. Further to create the subject index,} \]

\[ \frac{\text{graph size}}{c \times \text{initThreshold}} \geq \text{totalMem} \rightarrow \text{writeSPO} \hspace{1cm} (\because s = \text{graph size}) \hspace{1cm} (B.6) \]

where \text{writeSPO} \text{ is a method to create and update the subject, predicate and object indices. Similar to the previous case, } (s = \text{graph size}) > o \text{ because } s \text{ denotes the number of triples} \]
added to the graph and this equation is used after both, the predicate and object indices, are created. Each of the Equations B.4, B.5 and B.6 uses a user supplied constant $a$, $b$ and $c$ respectively. The following relationship holds between $a$, $b$ and $c$: $3 \geq a \geq b \geq c > 0$. Also, $a \leq 3$ since there are three in-memory data structures and, $a \geq b \geq c > 0$ so that triples can be added to other in-memory structures than the one currently under consideration in each equation. But, in a unified non-inference model, the subject index is not created, instead

$$dbThreshold = \frac{\text{graph size}}{c \times \text{initThreshold}}$$

(B.7)

is used to set up $dbThreshold$, which when reached a non-inference model switches to a relational database model. For the predicate and object indices, nodes are randomly selected from corresponding in-memory structures and all triples of that node are written to the respective index. The predicate, object and subject indices are created in that order, since Jena uses the subject, object and predicate in that order to execute a query.

As an example, an analysis of the impact of graph size along with totalMem for a non-inference model is presented using the above equations. Note that, initThreshold = $2.5 \times 10^6$ while $a = 3$, $b = 2.9$ and $c = 2.8$ in the following examples. When the graph size equals $5 \times 10^6$ triples and the totalMem equals 2, Equation B.4 is satisfied but Equations B.5 and B.6 are not satisfied, hence only the predicate index is created. On the other hand, if the graph size equals $15 \times 10^6$ triples and totalMem equals 2, all the equations are satisfied, however, a non-inference model switches to the database model.

From Figures B.13, B.15, it should be clear that with this new method of constructing a Lucene model (C-S-Eff: the third column from the left), a query time comparable to in-memory storage is achieved for SP2Bench queries, however, LUBM queries require a much longer querying time. For LUBM queries, the Pellet reasoner (Sirin et al., 2007) needs to query the larger predicate index multiple times, making it slower than the C-S approach, where the reasoner needs to query the relatively smaller subject and object indices. On the
other hand, $SP^2$Bench queries only use the in-memory subject and object structures, and hence perform as well as the in-memory model.

### B.5.6 Scalability results for Jena-MM

The studies conducted in previous subsections are now used to execute $SP^2$Bench and LUBM queries on a wide range of graph sizes. For experiments in this subsection, the algorithm used for $SP^2$Bench is degree centrality ($DC$) and the Lucene model construction strategy is C-S-Eff, while for LUBM the algorithm is MRU and the Lucene model construction strategy is C-S. For both, $SP^2$Bench and LUBM, when the number of triples, $T < 2.5 \times 10^6$, $initThreshold = 3/4 \times \text{graph size}$ and $totalMem = 1$, while for $T > 2.5 \times 10^6$, $initThreshold = 2.5 \times 10^6$ and $totalMem = 2$. For scalability tests, $initThreshold$ was set to these values as part of the test program. The test program uses queries Q3a and Q8 for $SP^2$Bench and queries Q6 and Q13 for LUBM, which were randomly selected from all available queries. An RDF stream was simulated by loading the largest dataset, $T = 15 \times 10^6$, and periodically querying the stream for various values of $T$ shown in the graphs.

Note that, in Figure B.16(b), the time for the SDB model is scaled down by a factor of 10, for $T = 1M$, for a good comparison. Further, in Figure B.16(c), for $T = 5M$ and $T = 15M$, the RDB model is scaled down by a factor of 15 and 30 while the SDB model is scaled down by a factor of 75 and 150, and the unified non-inference model is scaled down by a factor of 30 for $T = 15M$. Finally, in Figure B.16(d), for $T = 5M$ and $T = 15M$, the SDB model is scaled down by a factor of 35 and 7000 for a clear comparison. From Figures B.16(a) and B.16(b), it should be evident that the non-inference model performs as well as Jena’s in-memory model and performs better than Jena’s RDB and SDB models as the stream size increases. Also, Jena’s RDB and SDB models perform well for smaller sized streams but as the graph size increases their performance degrades. From Figures B.16(c) and B.16(d) one observes that for $T = 5 \times 10^6$, Jena’s in-memory model and the non-inference model give
Figure B.16. Comparison of SP²Bench Q3a and Q8 for various approaches (M denotes an Out Of Memory exception)

Figure B.17. Comparison of LUBM Q6 and Q13 for various approaches (M denotes an Out Of Memory exception)
almost similar results, better than Jena’s RDB and SDB models. For \( T = 15 \times 10^6 \), Jena’s in-memory model fails (denoted by an “M”), while the non-inference model gives results similar to Jena’s RDB model, since it has switched to the RDB model; these results are better than results obtained from the SDB model. The results presented for \( T = 5 \times 10^6 \) and \( T = 15 \times 10^6 \) can be directly correlated to the explanation following the equations given in Section B.5.5. Figure B.17 clearly shows that the unified inference model performs as well as Jena’s in-memory model for LUBM queries. Additionally, the inference model clearly outperforms Jena’s database backed models.
REFERENCES


VITA

Vaibhav Khadilkar was born in Mumbai, India. After completing his schoolwork at St. Mary’s High School in Mumbai in 2000, Vaibhav entered the University of Mumbai. He received a Bachelor of Engineering in Computer Science from The University of Mumbai in 2004. After that, he attended Lamar University in Beaumont, Texas between 2004-2006, where he received a Master of Science with a major in Computer Science. In January 2008, he entered the Doctoral program in the Department of Computer Science at The University of Texas at Dallas.