ABSTRACT

Title of dissertation: BUILDING EFFICIENT AND COST-EFFECTIVE CLOUD-BASED BIG DATA MANAGEMENT SYSTEMS

Abdul Hussain Quamar,
Doctor of Philosophy, 2015

Dissertation directed by: Professor Amol Deshpande
Department of Computer Science

In today's big data world, data is being produced in massive volumes, at great velocity and from a variety of different sources such as mobile devices, sensors, a plethora of small devices hooked to the internet (Internet of Things), social networks, communication networks and many others. Interactive querying and large-scale analytics are being increasingly used to derive value out of this big data. A large portion of this data is being stored and processed in the Cloud due the several advantages provided by the Cloud such as scalability, elasticity, availability, low cost of ownership and the overall economies of scale. There is thus, a growing need for large-scale cloud-based data management systems that can support real-time ingest, storage and processing of large volumes of heterogeneous data. However, in the pay-as-you-go Cloud environment, the cost of analytics can grow linearly with the time and resources required. Reducing the cost of data analytics in the Cloud thus remains a primary challenge. In my dissertation research, I have focused on building efficient and cost-effective cloud-based data management systems for different application domains that are predominant in cloud computing environments.

In the first part of my dissertation, I address the problem of reducing the cost of transactional workloads on relational databases to support database-as-a-service in the Cloud. The primary challenges in supporting such workloads include choosing how to partition the data across a large number of machines, minimizing the number of distributed transactions, providing high data availability, and tolerating failures gracefully. I have designed, built and evaluated SWORD, an end-to-end scalable online transaction processing system, that utilizes workload-aware data placement and replication to minimize the number of distributed transactions that incorporates a suite of novel techniques to significantly reduce the overheads incurred both during the initial placement of data, and during query execution at runtime.

In the second part of my dissertation, I focus on *sampling-based progressive analytics* as a means to reduce the cost of data analytics in the relational domain. Sampling has been traditionally used by data scientists to get progressive answers to complex analytical tasks over large volumes of data. Typically, this involves manually extracting samples of increasing data size (progressive samples) for exploratory querying. This provides the data scientists with user control, repeatable semantics, and result provenance. However, such solutions result in tedious workflows that preclude the reuse of work across samples. On the other hand, existing approximate query processing systems report early results, but do not offer the above benefits for complex ad-hoc queries. I propose a new progressive data-parallel computation framework, NOW!, that provides support for progressive analytics over big data. In particular, NOW! enables progressive relational (SQL) query support in the Cloud using unique progress semantics that allow efficient and deterministic query processing over samples providing meaningful early results and provenance

to data scientists. NOW! enables the provision of early results using significantly fewer resources thereby enabling a substantial reduction in the cost incurred during such analytics.

Finally, I propose NSCALE, a system for efficient and cost-effective complex analytics on large-scale graph-structured data in the Cloud. The system is based on the key observation that a wide range of complex analysis tasks over graph data require processing and reasoning about a large number of multi-hop neighborhoods or subgraphs in the graph; examples include *ego network* analysis, *motif counting* in biological networks, finding *social circles* in social networks, *personalized recommendations*, *link prediction*, etc. These tasks are not well served by existing vertex-centric graph processing frameworks whose computation and execution models limit the user program to directly access the state of a single vertex, resulting in high execution overheads. Further, the lack of support for extracting the relevant portions of the graph that are of interest to an analysis task and loading it onto distributed memory leads to poor scalability. NSCALE allows users to write programs at the level of neighborhoods or subgraphs rather than at the level of vertices, and to declaratively specify the subgraphs of interest. It enables the efficient distributed execution of these neighborhood-centric complex analysis tasks over large-scale graphs, while minimizing resource consumption and communication cost, thereby substantially reducing the overall cost of graph data analytics in the Cloud.

The results of our extensive experimental evaluation of these prototypes with several real-world data sets and applications validate the effectiveness of our techniques which provide orders-of-magnitude reductions in the overheads of distributed data querying and analysis in the Cloud.

BUILDING EFFICIENT AND COST-EFFECTIVE CLOUD-BASED BIG DATA MANAGEMENT SYSTEMS

by

Abdul Hussain Quamar

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor Amol Deshpande, Chair/Advisor
Professor Richard Marciano
Professor Jimmy Lin
Professor Alan Sussman
Professor Pete Keleher

Dedication

This dissertation is gratefully dedicated to

My loving wife Shahista Quamar

My mother Bano Quamar

My father Dr Masood Quamar and

My brother Abbas Quamar

Acknowledgments

Working as a PhD student at the University of Maryland at College Park has been a rewarding as well as a challenging experience for me that I shall forever cherish. This work would not have been possible without the invaluable educational, moral and physical support of many people. I owe my profound gratitude to the following people for helping me during the course go my PhD.

First and foremost, my deep gratitude goes to my advisor Prof Amol Deshpande for his unwavering support and mentorship throughout the course of my PhD. His constant encouragement and valuable guidance helped me immensely with my research work and in making steady progress throughout the years of my graduate work. I am deeply indebted to him for his gracious support.

I wish to profoundly thank Prof Jimmy Lin for his valuable guidance, cheerful enthusiasm and practical insights that helped me immensely in making progress towards my PhD. The resources provided by him for conducting the experimental evaluation of my work have been an invaluable support over the course of my dissertation research.

I would like to thank my committee members Prof Alan Sussman, Prof Pete Keleher and Prof Richard Marciano for agreeing to be on my dissertation committee and sparing their invaluable time for reviewing my thesis manuscript. My deep gratitude is also due to researchers Badrish Chandramouli and Jonathan Goldstein at Microsoft Research, who provided their valuable guidance and support during my research internships. These internships gave me an incredible opportunity for honing my system building skills that have been immensely valuable in making progress towards my PhD research.

I express my gratitude to Professors Aravind Srinivasan, Atif Memon, V.S. Subrahmanian, Nick Roussopoulos, Samir Khuller and Lise Getoor who have taught me during my graduate course work and advised me on my research work. The valuable insights and skills that I learnt from these esteemed professors have helped me immensely during the course of my research. I also wish to thank the chair, all the faculty and staff of the Department of Computer Science especially Jennifer Story and Fatima Bangura for their valuable administrative support during the past several years. I would also like to thank the UMIACS staff who have provided valuable technical support for the resources that I have used for conducting the experimental evaluation of my research.

My special gratitude goes to my colleagues in the computer science department and especially in the database group whose help and collaboration have been immensely useful in making progress towards my PhD over the past several years. I would like to make a special mention for Ashwin K. Kayyoor, Souvik Bhattacharjee, Amit Chavan, Udayan Khurana, Theodoris Rekatsinas, Walaa Eldin Moustafa, Hui Miao, Jayanta Mondal, Mossaab Bagdouri, Ben London, Ahmed E. Kosba, Karla Saur and Ioana Bercea for their valuable help and support. I have immensely enjoyed working, collaborating and interacting with these people throughout my PhD.

I owe my deepest gratitude to my family especially my wife, my parents, my kids and my brother for their constant support and belief in me during the course of my PhD. No words can express the profound gratitude that I owe to them. This thesis would not have been possible without their support.

Last but not the least I wish to thank the Almighty God for his extreme benevolence and graciousness and everything that he has provided for.

Portions of this dissertation research have been generously supported by the NSF grant CCF-0937865, NSF Grant 1319432, an IBM Collaborative Research Award, and an Amazon AWS in Education Research grant.

Table of Contents

List of Tables

List of Figures

‘

Chapter 1: Introduction

1.1 Cloud-Based Big Data Management Systems

Over the last decade there has been an explosive growth in data that is being generated by a wide variety of sources. These include data being produced by information networks; a variety of static and mobile sensors; mobile devices; social media interactions on forums and online platforms such as Twitter, Facebook, etc.; logs produced by systems and devices; data generated by cloud applications and geo-distributed services, etc. A wide variety of businesses and application domains such as marketing and advertisement, financial services, healthcare, retail, telecommunications, gaming, etc., can benefit from the querying and analysis of this big data to derive meaningful value for solving challenging real-world problems. Examples include gaining useful business insights from customer trends, profiles and activities; prediction of security threats and vulnerabilities; fraud detection and prevention; operational analysis of logs, sensor and machine data for improved corporate decision making, etc. There is thus, a growing need for data management systems that can support real-time ingest, storage, integration, interactive querying and complex analysis over large volumes of heterogeneous data coming from different sources.

A large portion of this massive amount of data is increasingly being stored and

processed in the Cloud due to economies of scale. Processing data in the Cloud provides several advantages such as *scalability* to store, manage and process large volumes of data; *elasticity* to enable utilization of resources as per requirement; *availability* to minimize application downtimes; *low cost of ownership* enabling small business to take advantage of the cloud, etc. These significant advantages have led to the rapid growth of several cloud-based storage and compute services such as Amazon's EC-2, AWS, S3; Google's compute cloud; Dropbox; Rackspace and many others. A host of services and applications such as document/spreadsheet services, web-based email, social networking platforms (e.g. Facebook, Twitter, LinkedIn etc.) content distribution (e.g. Netflix, Amazon Prime); etc., all live in and benefit from the Cloud today. The pay-as-you-go paradigm of the Cloud causes computation costs to increase linearly with query execution time and the amount of resources consumed, which in turn depend on the size of the data and the complexity of the analytics performed. Reducing the cost of interactive querying and complex analytics over large volumes of data in the Cloud while maintaining the desired latency, quality and efficacy of analysis results, thus remains a primary challenge for large-scale cloud-based data processing systems today.

My dissertation research focuses on building data management systems that can efficiently execute interactive queries and complex analytics over large-scale data while minimizing the cost of such data processing. Pursuant to this direction, I have focused on two different approaches to reduce the cost of data processing in the Cloud. The first approach concentrates on building algorithms and techniques that reduce the memory requirements, communication and I/O overheads and complexity of distributed computation. In essence, it allows the system to consume fewer resources while providing the

same or better service. The second approach uses progressive analysis techniques to reduce the cost by doing less work. It enables the production of results over partial data and refinement of these results as more data is consumed. It also enables users to stop computations early if sufficient accuracy is reached or query incorrectness is observed.

More specifically, I have designed, built and evaluated three different data processing systems which address the challenges for three different application domains that are predominant in cloud computing environments today. These are: (1) Transactional applications that require ACID guarantees, (2) Applications that involve analytics over data in the traditional relational domain, and (3) Complex analytics over large-scale graph-structured data. I further elaborate on these in the following sections.

1.2 Scaling Transactional Applications

A large number of cloud-based applications that query data stored in traditional relational tables require transactional guarantees. Transparently scaling such OLTP workloads to support database-as-a-service requires mechanisms to partition the data onto multiple machines and provide high data availability. Horizontal partitioning (or sharding) is a technique used to partition the data onto multiple machines, wherein rows of the relational tables are divided and placed into separate partitions that are spread across different machines. The commonly used techniques for horizontal partitioning include hash-based partitioning [1], round robin partitioning, and range partitioning. Replication is a technique wherein multiple copies of a data item are created and placed on separate partitions to provide high availability. These techniques are now routinely used to store, query,

and analyze very large datasets, and have become an integral part of any large-scale data management system that supports transactional applications.

A natural consequence of employing sharding and/or replication on transactional workloads is that transactions or queries may need to access data from multiple partitions. This is usually not a problem for analytical workloads where this is, in fact, desired and can be exploited to parallelize the query execution itself. However, to ensure transactional semantics, distributed *transactions* must employ a distributed consensus protocol (e.g., 2-phase commit or Paxos commit [2]), which can result in high and often unacceptable latencies [3] and consequently a high cost of executing the workload in the Cloud environment. During the last decade, this has led to the emergence and wide use of key-value stores that do not typically support transactional consistency, or often restrict their attention to simple single-item transactions.

Over the last few years, there has been an increasing realization that the functionality and guarantees offered by key-value stores are not sufficient in many cases, and there are many ongoing attempts to scale out OLTP workloads without compromising the ACID guarantees. H-Store [4] is an attempt to rethink OLTP query processing by using a distributed main memory database, but requires that the transactions be pre-defined in terms of stored procedures and not span multiple partitions. Google's Megastore [5] provides ACID guarantees within data partitions but limited consistency guarantees across them and has poor write throughput. Moreover, the database features provided by Megastore are limited to the semantics that their partitioning scheme can support.

Capturing and modeling the transactional workload over a period of time, and then exploiting that information for data placement and replication has been shown [6] to pro-

vide significant benefits in performance, both in terms of transaction latencies and overall throughput. However, such workload-aware data placement approaches are not scalable as they can incur very high partitioning and routing (dispatching transactions to appropriate partitions) overheads. Further, they do not deal with workload changes and may perform worse than naive approaches even for small workload changes. Thus, efficiently supporting ACID guarantees for transactions while employing sharding and replication for scalability and availability, remains a challenge in this environment.

1.3 Big Data Analytics in the Relational Domain

Complex analytics on large-scale data for applications in the relational domain involves processing of data by relational operators using a relational query processing engine. These analytical tasks when carried out in the Cloud can be quite expensive depending upon the amount of resources required. The problem is exacerbated by the exploratory nature of such analytics, where queries are iteratively discovered and refined, and include the submission of many off-target and erroneous queries (e.g., bad parameters). In traditional systems, queries must execute to completion before such problems are diagnosed, often after hours of expensive compute time are used up. In order to reduce the cost of such analytics in the Cloud, data scientists typically choose to perform their ad-hoc querying on extracted *samples* of data. This approach gives them the control to carefully choose from a huge variety [7–9] of sampling strategies in a domain-specific manner. For a given sample, it provides precise (e.g., relational) query semantics, repeatable execution using a query processor and optimizer, result provenance in terms of what data contributed to an

observed result, and query composability. Further, since choosing a fixed sample size a priori for all queries is impractical, data scientists usually create and operate over multiple *progressive samples* of increasing size [8].

In an attempt to help data scientists, the database community has proposed *approximate query processing* (*AQP*) systems such as CONTROL [10] and DBO [11] that perform *progressive analytics*. We define progressive analytics as the generation of early results to analytical queries based on partial data, and the progressive refinement of these results as more data is received. Progressive analytics allows users to get early results using significantly fewer resources, and potentially end (and possibly refine) computations early once sufficient accuracy or query incorrectness is observed.

The general focus of AQP systems has, however, been on automatically providing confidence intervals for results, and selecting processing orders to reduce bias [12–16]. The premise of AQP systems is that users are not involved in specifying the semantics of early results; rather, the system takes up the responsibility of defining and providing accurate early results. To be useful, the system needs to automatically select effective sampling strategies for a particular combination of query and data. This can work for narrow classes of workloads, but does not generalize to complex ad-hoc queries. Further, traditional scalable distributed frameworks such as Map Reduce (MR) are not pipelined, making them unsuitable for progressive analytics. Map Reduce Online (MRO) [17] adds pipelining, but does not offer the semantic underpinnings of progress necessary to achieve the desirable features outlined above.

To summarize, data scientists prefer user-controlled progressive sampling because it helps avoid the above issues, but the lack of system support results in a tedious and

error-prone workflow that precludes the reuse of work across progressive samples. We need a system that (1) allows users to communicate progressive samples to the system; (2) allows efficient and deterministic query processing over progressive samples, without the system *itself* trying to reason about specific sampling strategies or confidence estimation; and yet (3) continues to offer the desirable features outlined above at scale.

1.4 Data Analytics on Large-Scale Graphs

Over the last decade, information networks have become ubiquitous and widespread. These include social networks, communication networks, Web, financial transaction networks, citation networks, gene regulatory networks, disease transmission networks, ecological food networks, sensor networks, RDF knowledge bases, and more. Network data also arises in applications like phone call data, IP traffic data, health-care data, source code repositories, parcel shipment data, and so on. Social contact graphs are expected to be available for analysis in near future, and can potentially be used to gain insights into various social phenomena as well as disease outbreak and prevention. The ubiquity of these information networks has led to an unprecedented growth of graph-structured data, since representing information network data as a graph is most natural; with nodes representing the entities and edges denoting the interactions between them.

The domain of data analytics on such large-scale graph structured data is growing rapidly. There is increasing interest in applications that require executing complex analytics over such graph data to get valuable insights into the network's functional abilities, for scientific discovery, for event or anomaly detection, for assessment of potential impact of

interventions, etc. Many of these complex analysis tasks on graphs require processing a large number of multi-hop neighborhoods or subgraphs. Some specific examples include ego network analysis, motif counting, finding social circles, personalized recommendations, link prediction, anomaly detection, analyzing influence cascades, and others. Although there have been quite a few *vertex-centric* graph processing frameworks proposed in recent years, these tasks are typically not well-served by those. This is because, in those frameworks, user programs are only able to directly access the state of a single vertex at a time, resulting in high communication, scheduling, and memory overheads in executing such tasks.

Further, most existing graph processing frameworks ignore the challenges in extracting the relevant portions of the graph that an analysis task is interested in, and loading those onto distributed memory. In many cases, the user may only want to analyze a subgraph (or several subgraphs) of the overall graph that would require access to a subset of the nodes and edges. Naively loading each disk partition of the graph onto a separate machine may lead to unnecessary distributed communication, especially for distributed graph analytics, where the number of messages exchanged typically increases super-linearly with the number of machines used. This is likely to become a scalability bottleneck especially for subgraph-centric analysis tasks.

As such, we see that the existing vertex centric approaches are good at certain graph computation tasks such as computing Page Rank, Connected components, etc., which do not require aggregating neighbor state information. These approaches do not scale well to large graphs for subgraph-centric graph analysis tasks because of fundamental limitations imposed by their computation and execution models as mentioned above.

1.5 Thesis Contributions

In this section I provide an overview of the contributions of my dissertation research that provides solutions for building cost effective systems for data processing in the Cloud for the above mentioned application domains.

1.5.1 Scalable Workload-Aware Data Placement for Transactional Workloads

To address the problem of scaling transactional workloads to provide Database-as-a-Service in the Cloud, we built SWORD [18, 19], a *scalable workload-aware* data partitioning and placement approach for transparently scaling out standard OLTP workloads with full ACID support. Our key contributions in this work are a suite of novel techniques to achieve higher scalability, and to increase tolerance to failures and to workload changes. We model the workload as a hypergraph over the data items, and propose using a *hypergraph compression* technique to reduce the overheads of partitioning. To deal with workload changes, we propose an incremental data repartitioning technique that modifies data placement in small steps without resorting to complete workload repartitioning. We have built a workload-aware *active replication* mechanism in SWORD to increase availability and enable load balancing. We propose the use of *fine-grained quorums* defined at the level of *groups of tuples* to control the cost of distributed updates, improve throughput, and provide adaptability to different workloads. To our knowledge, SWORD is the first system that uses fine-grained quorums in this context. Summarizing,

the major contributions of this work are:

- *Effective workload modeling and compression* that reduces partitioning and book-keeping overheads, and enables handling of both new tuples and those not represented in the workload.
- *Incremental repartitioning* to mitigate performance degradation due to workload changes without a complete repartitioning.
- *Use of fine-grained quorums* to control the cost of distributed updates, to improve throughput, and to cater to OLTP workloads with a mix of different access patterns.
- *Workload-aware replication* mechanism that attempts to *disentangle* conflicting transactions leading to better data placement.
- *Efficient and scalable routing mechanism* that minimizes the number of partitions to involve for a given query and uses compact routing tables to minimize memory requirements.

The results of our experimental evaluation on SWORD deployed on an Amazon EC2 cluster show that our techniques result in orders-of-magnitude reductions in the partitioning and book-keeping overheads, and improve tolerance to failures and workload changes; we also show that choosing quorums based on the query access patterns enables us to better handle query workloads with different read and write access patterns.

1.5.2 Progressive Analytics on Big Data in the Cloud

To address the problem of reducing the cost of big data analytics in the Cloud we propose a new *progressive analytics* system called NOW! based on a progress model

called PRISM that (1) allows users to communicate progressive samples to the system; (2) allows efficient and deterministic query processing over samples; and (3) provides repeatable semantics and provenance to data scientists. Instead of modifying an existing relational engine to support progressive analytics, we use an unmodified temporal streaming engine, by carefully reinterpreting its temporal fields to denote progress.. Based on PRISM (Section 4.2), we have built NOW!, a progressive data-parallel computation framework on the Azure cloud platform, where progress is understood as a first-class citizen in the framework. NOW! generalizes the popular data-parallel Map Reduce (MR) model and supports *progress-aware reducers* that understand explicit progress in the data. NOW! works with these "progress-aware reducers"– in particular, it works with streaming engines to support progressive SQL over big data.

I summarize the major contributions in this work below:

- We designed and built NOW!, a new pipelined computation platform for the Cloud, that natively supports progressive queries with explicit progress semantics built into the framework as a fundamental building block. NOW! has several important features which are summarized below:
  - Fully pipelined progressive computation and data movement across multiple stages with different partitioning keys, in order to avoid the high cost of sending intermediate results to cloud storage.
  - Elimination of sorting in the framework using progress-ordered data movement, partitioned computation pushed inside progress-aware reducers, and support for the traditional reducer API.

- Progress-based merge of multiple map outputs at a reducer node.
- Concurrent scheduling of multi-stage map and reduce jobs with a new scheduling policy and flow control scheme.

- We show that an *unmodified* streaming engine with temporal semantics (such as Microsoft StreamInsight) can be used to answer SQL queries with progress semantics.

- We extend NOW! with features for high performance and minimizing resource utilization and provide mechanisms for elasticity and migration .

- We provide a detailed evaluation with real and synthetic datasets (up to 100GB) on NOW! used with StreamInsight on Windows Azure.

Extensive experiments on Windows Azure with real and synthetic workloads validate the scalability and benefits of NOW! and its optimizations, over current solutions for progressive analytics.

1.5.3 Neighborhood-centric Analytics on Large-scale Graphs in the Cloud

For cost-effective complex analytics on graph-structured data we propose NSCALE [20–22], a novel end-to-end graph processing framework aimed at supporting complex *subgraph-centric* analytics over large-scale graphs in the cloud. NSCALE enables users to write programs at the level of subgraphs rather than at the level of vertices. Unlike most previous graph processing frameworks, which apply the user program to the entire graph, NSCALE allows users to declaratively specify subgraphs of interest. Our framework in-

cludes a novel graph extraction and packing (GEP) module that utilizes a cost-based optimizer to partition and pack the subgraphs of interest into memory on as few machines as possible. The distributed execution engine then takes over and runs the user program in parallel on those subgraphs, restricting the scope of the execution appropriately, and utilizes novel techniques to minimize memory consumption by exploiting overlaps among the subgraphs. Our key contributions in this work are summarized below:

- **Subgraph-centric programming model.** Unlike vertex-centric frameworks, NSCALE allows users to write custom programs that access the state of entire *subgraphs* of the complete graph. This model is more natural and intuitive for many complex graph analysis tasks compared to the popular vertex-centric model.

- **Extraction of query subgraphs.** Unlike existing graph processing frameworks, most of which apply user programs to the entire graph, NSCALE efficiently supports tasks that involve only a select set of subgraphs (and of course, NSCALE can execute programs on the entire graph if desired).

- **Efficient packing of query subgraphs.** To enable efficient execution, subgraphs of interest are packed into as few containers (i.e., memory) as possible by taking advantage of overlaps between subgraphs. The user is able to control resource allocation (for example, by specifying the container size), which makes our framework highly amenable to execution in cloud environments.

- **Support for iterative analysis tasks.** NSCALE supports the Bulk Synchronous Protocol (BSP) model for executing iterative analysis tasks like computation of PageRank or global connected components. NSCALE's BSP implementation is most similar to

that of GraphLab, and the information exchange is achieved through shared state updates between subgraphs on the same partition and through use of "ghost" vertices (i.e., replicas) and message passing between subgraphs across different partitions.

NSCALE has been implemented and deployed over two data distribution and computation frameworks. The first is the popular Apache YARN framework and the second is the Apache Spark framework. Both these frameworks/platforms have been extensively used for big data processing and thus were a natural choice for deploying NSCALE. We present a comprehensive empirical evaluation comparing against three state-of-the-art systems, namely, Giraph, GraphLab, and GraphX, on several real-world datasets and a variety of analysis tasks. Our experimental results show orders-of-magnitude improvements in performance and drastic reductions in the cost of analytics compared to vertex-centric approaches.

1.6 Organization

In Chapter 2 we review work related to workload-aware data placement, progressive analytics on big data and graph analytics on large-scale graph-structured data. In Chapter 3 we discuss in detail the concept of workload modeling and present the design and architecture of our system SWORD. In Chapter 4 we provide a background on PRISM, the progress model used by NOW!. We then present the detailed design and architecture of NOW! which enables progressive analytics on big data in the Cloud. In Chapter 5 we describe the design and architecture of NSCALE, our proposed framework for neighborhood-centric analytics on large-scale graphs in the Cloud. Finally in Chap-

ter 6, we provide our concluding remarks on building cost effective cloud-based data processing systems and briefly discuss directions for future work.

Chapter 2: Related Work

In this chapter I first discuss the related work in the area of workload-aware data placement. I then review other related work in area of progressive analytics on big data and graph analytics on large-scale graph structured data.

2.1 Scaling Transactional Workloads

As distributed databases have grown in scale, partitioning and data replication to minimize overheads and improve scalability has received a lot of interest.

2.1.1 Workload-aware Data Placement

Among workload-aware data placement techniques, Schism [6], is closest to our work. It uses a schema-independent approach which observes and captures the query and transaction workload over a period of time, and utilizes this workload information to achieve a data placement that minimizes the number of distributed transactions. Their approach models the transaction workload as a graph over the database tuples, where an edge indicates that the two tuples it connects appear together in a transaction; it then uses a graph partitioning algorithm to find a data placement that minimizes the number of distributed transactions thus increasing throughput significantly over baseline approaches.

There are significant differences between Schism and SWORD in handling the critical issues of scalability, availability, fault-tolerance and dealing with workload changes. Schism does not provide any mechanism to deal with workload changes. Another difference between our framework and Schism is the use of aggressive replication. Schism trades off performance for fault tolerance by not replicating data items with a high write/update frequency. This might compromise the availability of these data items in presence of failure and affect the ability to do load balancing across multiple machines. We instead replicate each tuple at least once, and possibly more times depending on the access frequencies. The replicas are kept strongly consistent. We also empirically show that our approach is more fault-tolerant than tuple-level fine-grained partitioning techniques such as Schism described above.

In a follow-up work to Schism, Tatarowicz et al. [23] use a powerful router with high memory and computational resources and employ compression to scale up the lookup tables. However, that approach suffers from the same issues as a scaled-up architecture and may not be cost effective because it needs large memory and computation resources. In our work, we minimize the lookup table sizes significantly by using a compressed representation of the workload.

Workload-aware approaches have also been used in the past (e.g., AutoAdmin project [24]) for tuning the physical database design, i.e., identifying the physical design structures such as indexes for a given database and workload. Kumar et al. [25] propose a workload-aware approach for data placement and co-location for read-only analytical workloads. The solutions proposed in that paper focus on optimizing the energy and resource consumption, unlike our work that deals with data placement for minimizing

distributed transactions for OLTP workloads. Pavlo et al. [26] propose a workload-aware approach for automatic database partitioning for OLTP applications. However, their approach does not provide an incremental mechanism to deal with workload changes once data is partitioned.

The partitioning strategies in cloud/NoSQL systems [27, 28] primarily aim for scalability by compromising the consistency guarantees. Moreover, the partitioning in [27] cannot be changed without reloading the entire dataset. On the other hand, we endeavor to scale OLTP workloads while maintaining transactional ACID properties and provide an incremental repartitioning mechanism to deal with workload changes.

Nehme et al. [29] have developed a system to automatically partition the database based on the expected workload. Their approach is tightly integrated with the query optimizer which relies on database statistics. However their approach ignores the structural and access correlations between queries that we consider by modeling the query workload as a hypergraph.

2.1.2 Replication

Replication has been widely used in distributed databases for availability, load balancing and fault tolerance [30–32]. In this dissertation, we focus on active replication as it provides increased availability and load balancing. Gray et al. [33] showed that replication in distributed databases can result in performance bottlenecks and can limit their scalability. We address the performance issues related to replication by using a workload-aware replication technique which further minimizes distributed transactions with the use

of in-graph replication and control update costs using fine-grained quorums. Although quorums have been used to alleviate the cost of replica updates [34], we have shown that a static choice of quorums for all data items is not sufficient to handle different workloads with varying access patterns.

2.2 Progressive Analytics on Big Data in the Cloud

There is a substantial body of prior work in the area of progressive analytics which can be grouped into four broad categories: (1) Approximate query processing (AQP), (2) Map-Reduce framework variants, (3) Distributed stream processing systems and (4) Interactive Full-Data Analytics. I review related work for each of these below.

2.2.1 Approximate Query Processing

Online aggregation was originally proposed by Hellerstein et al. [12], where the focus was on grouped aggregation with statistically robust confidence intervals based on random sampling. This was extended to handle join queries using the ripple join [15] family of operators. Specialized sampling techniques have been widely studied in subsequent years (e.g., see [10, 13, 35]). Laptev et al. [36] propose iteratively computing MR jobs on increasing data samples until a desired approximation goal is achieved. BlinkDB [37] constructs a large number of multi-dimensional samples offline using a particular sampling technique (stratified sampling) and chooses samples automatically based on a user-specified budget.

We follow a different approach: instead of the system taking responsibility for query

accuracy (e.g., as sampling techniques) which may not be possible in general, we involve the user (the query writer) in the specification of progress semantics. A query processor using PRISM can support a variety of user-defined progressive sampling schemes; we view prior work described above as part of a layer between our generic progress engine and the user, that helps with the assignment of PIs in a semantically appropriate manner.

2.2.2 MR Framework Variants

Map-Reduce Online *(MRO)* [17] supports progressive output by adding pipelining to MR. Early result snapshots are produced by reducers, each annotated with a rough progress estimate based on averaging progress scores from different map tasks. Unlike our techniques, progress in MRO is an operational and non-deterministic metric that cannot be controlled by users or used to formally correlate progress to query accuracy or to specific input samples. From a data processing standpoint, unlike NOW!, MRO sorts subsets of data by key and can incur redundant computations as reducers repeat aggregations over increasing subsets (see [38] for more details).

Li et al. [39] propose scalable one pass analytics (SOPA), where they replace sort-merge in MR with a hash-based grouping mechanism inside the framework. Our focus is on *progressive* queries, with a goal of establishing and propagating explicit progress in the platform. Like SOPA, we eliminate sorting in the framework, but leave it to the reducer to process progress-sync ordered data. Streaming engines use efficient hash-based grouping, allowing us to realize similar performance gains as SOPA inside our reducers.

2.2.3 Distributed Stream Processing

Stream processing engines (SPEs) answer real-time temporal queries over windowed streams of data. We tackle a different problem: progressive results for atemporal queries over atemporal offline data, and show that our new progress model can in fact be realized by leveraging and re-interpreting the notion of time used by temporal SPEs. NOW! is an MR-style distributed framework for progressive queries; it is markedly different from distributed SPEs [40] as it leverages the explicit notion of progress to build a batched-sequential data-parallel framework that does not target real-time data or low-latency queries. The use of progress-batched files for data movement allows NOW! to amortize transfer costs across reducer per-tuple computation cost. NOW!'s architecture is designed along the lines of MR with extended map and reduce APIs, and is designed for the Cloud.

2.2.4 Interactive Full-Data Analytics

Dremel [41] and PowerDrill [42] are distributed systems for interactive analysis of read-only large columnar datasets. Spark [43] provides in-memory data structures to persist intermediate results in memory, and can be used to interactively query big data sets or get medium-latency batch-wise results on real-time data [44]. These engines have a different goal from us; by fully committing memory and compute resources a priori, they provide full results to queries on hot in-memory data in milliseconds, for which they use careful techniques such as columnar in-memory data organization for the (smaller) subset of data that needs such interactivity. On the other hand, we provide generic interactivity

over large datasets, in terms of early meaningful results on progressive samples and refining results as more data is processed. Based on the early results, users can choose to potentially end (or possibly refine) computations once sufficient accuracy or query incorrectness is observed.

2.3 Data Analytics on Large-scale Graph Structured Data

In this section we focus related work in the area on the large-scale graph processing frameworks and programming models.

2.3.1 Vertex-centric Approaches.

Most existing graph processing frameworks such as Pregel [45], Apache Giraph, GraphLab [46], Kineograph [47], GPS [48], Grace [49], etc., are vertex-centric. Users write *vertex-level programs*, which are then executed by the framework in either a bulk synchronous fashion (Pregel, Giraph) or asynchronous fashion (GraphLab) using message passing or shared memory. These frameworks fundamentally limit the user program's access to a single vertex's state – in most cases to the local state of the vertex and its edges. This is a serious limitation for many complex analytics tasks that require access to subgraphs.

For example, to analyze a 2-hop neighborhood around a vertex to find social circles [50], one would first need to gather all the information from the 2-hop neighbors through message-passing, and reconstruct those neighborhoods locally (i.e., in the vertex program local state). Even something as simple as computing the number of triangles

for a node requires gathering information from 1-hop neighbors (since we need to reason about the edges between the neighbors). This requires significant network communication and an enormous amount of memory. Consider some back-of-the-envelope calculations for estimating the message passing and memory overhead for constructing neighborhoods of various sizes at each vertex for the Orkut social network graph with approx 3M nodes, 234M edges and an average degree of 77. The original graph occupies 14GB of memory for a data structure that stores the graph as a bag of vertices in adjacency list format. Table 2.1 provides an estimate of the number of messages that would need to be exchanged and the memory footprints required in order to construct 1- and 2-hop neighborhoods at each vertex for ego network analysis. It is clear that a vertex-centric approach requires inordinate amounts of network traffic, beyond what can be addressed by "combiners" in Pregel [45] or GPS [48], and impractical amount of cluster memory. Although GraphLab is based on a shared memory model, it too would require two phases of GAS (Gather, Apply, Scatter) to construct a 2-hop neighborhood at each vertex and suffers from duplication of state and high memory overhead.

We also see that even for a modest graph, the memory requirements are quite high for most clusters today. Furthermore, because most existing graph processing frameworks hash-partition vertices by default, this approach will create much duplication of neighborhood data structures. In recent work, Seo et al. [51] also observe that these frameworks quickly run out of memory and do not scale for ego-centric analysis tasks.

The other weakness of existing vertex-centric approaches is that they almost always process the entire graph. In many cases, the user may only want to analyze a subset of the subgraphs in a large graph (for example, focusing in only on the neighborhoods sur-

Neighborhood size	1-Hop	2-Hop
Messages required to construct neighborhoods	231 M	$\approx$ 18 B
Avg. Memory required per neighborhood	83 KB	6 MB
Total Cluster Memory required	233 GB	$\approx$ 18 TB

Table 2.1: **Message passing and memory overheads of an vertex-centric approach, for constructing neighborhoods of different sizes at each vertex for executing an ego-centric analysis task (the input Orkut graph has 3M nodes and 234M edges).**

rounding "persons of interest" in a social network, or only the subgraphs induced by a set of "hashtags" depicting current events in the Twitter network). Naively loading each partition of the graph onto a separate machine may lead to unnecessary network communication, especially since the number of messages exchanged increases non-linearly with the number of machines.

2.3.2 Existing Subgraph-centric Approaches.

While researchers have proposed a few subgraph-centric frameworks such as Giraph++ [52] and GoFFish [53], there are significant limitations associated with both. These approaches primarily target the message passing overheads and scalability issues in the vertex-centric, BSP model of computation. Giraph++ partitions the graph onto multiple machines, and runs a sequential algorithm on the entire subgraph in a partition in each superstep. GoFFish is very similar and partitions the graph using METIS (another scalability issue) and runs a connected components algorithm in each partition. An important distinction is that in both cases, the subgraphs are determined by the system, in contrast to our framework, which explicitly allows users to specify the subgraphs of interest. Furthermore, these previous frameworks use serial execution within a partition and the onus of parallelization is left to the user. It would be extremely difficult for the end

user to incorporate tools and libraries to parallelize these sequential algorithms to exploit powerful multicore architectures available today.

2.3.3 Other Graph Processing Frameworks.

There are several other graph programming frameworks that have been recently proposed. SociaLite [54] describes an extension of a Datalog-based query language to express graph computations such as PageRank, connected components, shortest path, etc. The system uses an underlying relational database with tail-nested tables and enables users to hint at the execution order. Galois [55], LFGraph [56], are among highly scalable general-purpose graph processing frameworks that target systems- or hardware-level optimization issues, but support only low-level or vertex-centric programming frameworks. Facebook's Unicorn system [57] constructs a distributed inverted index and supports online graph-based searches using a programming API that allows users to compose queries using set operations like AND, OR, etc.; thus Unicorn is similar to an online SPARQL query processing system and can be used to identify nodes or entities that satisfy certain conditions, but it is not a general-purpose complex graph analytics system.

X-Stream [58] provides an edge-centric graph processing model using streamed partitions on a single shared memory machine. The programming API is based on scatter and gather functions that are executed on the edges and that update the states maintained in the vertices. Any multi-hop traversal in X-Stream would be expensive as it requires multiple iterations of the scatter, shuffle and gather phases. Since the stream partitioning used by the framework does not take the neighborhood structure into account, such

operations would necessitate a large amount of data to be shuffled to the gather phase across different stream partitions. X-Stream also fundamentally relies on the vertex state remaining constant in size, and it would negate the key benefits of X-Stream if variable-sized neighborhoods were constructed in the vertex state. Finally, X-Stream provides a restricted edge-centric API that would make it hard to encode neighborhood-centric computations such as those supported by NSCALE.

GraphX, built on top of Apache Spark, supports a flexible set of operations on large graphs [59]; however, GraphX stores the vertex information and edge information as separate RDDs, which necessitates a join operation for each edge traversal. Further, the only way to support subgraph-centric operations in GraphX is through its emulation of the vertex-centric programming framework, and our experimental comparisons with GraphX show that it suffers from the same limitations of the vertex-centric frameworks as discussed above.

Chapter 3: Scalable Workload-aware Data Placement for Transactional Workloads

3.1 Introduction

In this chapter, we discuss our work on the problem of transparently scaling out transactional (OLTP) workloads on relational databases, to support *database-as-a-service* in cloud computing environment. The primary challenges in supporting such workloads include choosing how to *partition* the data across a large number of machines, minimizing the number of *distributed transactions*, providing high data *availability*, and tolerating *failures* gracefully. Workload-aware data placement and replication approaches such as the schema-independent approach proposed by Curino et al. [6] have been shown to provide significant benefits in performance, both in terms of transaction latencies and overall throughput. However, such workload-aware data placement approaches can incur very high partition and execution overheads, and further, may perform worse than naive approaches if the workload changes. There are several challenges in employing a fine-grained workload-aware data placement approach such as Schism. We highlight these below:

Scalability: The initial cost of partitioning and the follow-on cost of maintaining the

partitions can be very high, and in fact, it is unlikely that the fine-grained partitioning approach can scale to really large data volumes;

Efficient Routing: The routing tables that store the tuple-to-partition mappings, required to dispatch the queries or transactions to appropriate partitions, can become very large and expensive to consult;

Workload Modeling: It is not clear how to handle newly inserted tuples, or tuples that do not appear in the workload;

Dealing with Workload Change: The performance for such an approach can be worse than random partitioning if the workload changes significantly; a consequence of overfitting the data placement to a particular workload.

Load balancing and Handling Failures: Random hash-based partitioning schemes often naturally have better load balancing and better tolerance to failures as compared to a fine-grained partitioning approach.

In this dissertation, we propose SWORD, a scalable workload-aware data partitioning and placement approach for OLTP workloads, that incorporates a suite of novel techniques to significantly reduce the overheads incurred both during the initial placement, and during query execution at runtime. We elaborate on the key ideas proposed in SWORD below.

Hypergraph compression to reduce book-keeping overheads: We model the workload as a *hypergraph*[1], where each *hyperedge* corresponds to a transaction or a query[2], and em-

[1]We chose a hypergraph-based representation because we observed better performance, but we could also use a graph-based representation instead.

[2]Hereafter we use the term *transaction* to denote both an update transaction or a read-only query.

ploy hypergraph partitioning algorithms to guide data placement decisions. We propose using a *two-phase approach*, where we first *compress* the hypergraph using either hash partitioning or an analogous simple and easy-to-compute function, and then *partition* the compressed hypergraph. This results in a substantial reduction in the sizes of the mapping tables required for dispatching the transactions to appropriate partitions. As we show, this simple hybrid approach is able to reap most of the benefits of fine-grained partitioning at a much lower cost, resulting in significantly higher throughputs. Our approach also naturally handles both new tuples and tuples that were not accessed in the workload. Further, it is able to deal with changes in workload more gracefully and is more effective at tolerating failures.

Incremental Repartitioning: We propose an *incremental repartitioning* technique to deal with workload changes, that monitors the workload to identify significant changes, and repartitions the data in small steps to maintain a good overall partitioning. Our approach is based on efficiently identifying *candidate* sets of data items whose migration has the potential to reduce the frequency of distributed transactions the most, and then performing the migrations during periods of low load.

Fine-Grained Quorums: Third, we propose using *fine-grained quorums* to alleviate the cost of distributed updates for active replication and to gracefully deal with partition failures. Unlike prior work [33, 34] where the types of quorum are chosen a priori and uniformly for all data items, we choose the type of quorum independently for groups of tuples based on their combined read/write access patterns. This allows us to cater to typical OLTP workloads that have a mix of read and write queries with varying access

patterns for different data items.

Aggressive Replication: We propose an aggressive replication mechanism that attempts to *disentangle* conflicting transactions through replication, enabling better data placement.

Scalable Routing mechanism: Finally, we develop an efficient *query routing* mechanism to identify which partitions to involve in a given transaction. Use of aggressive replication and quorums makes this very challenging, and in fact, the problem of identifying a minimal set of partitions for a given query is a generalization of the *set cover* problem (which is not only NP-Hard but also hard to approximate). We develop a greedy heuristic to solve this problem. We also develop a compact routing mechanism that minimizes memory overheads and improves lookup efficiency.

Our experimental evaluation of SWORD deployed on an Amazon EC2 cluster demonstrates that our hypergraph-based workload representation and use of in-graph replication based on access patterns, lead to a much better quality data placement as compared to other data placement techniques. We show that our scaling techniques result in orders-of-magnitude reductions in the partitioning overheads including the workload partitioning time, cost of distributed transactions, and query routing times for data sets consisting of up to a billion tuples. Our incremental repartitioning technique effectively deals with the performance degradation caused by workload changes using minimal data movement. We also show that our techniques provide graceful tolerance to partition failures compared to other data placement techniques.

3.2 SWORD Overview

We begin with providing a high-level overview of SWORD's architecture. We then briefly present the basic workload-aware approach that captures a transaction workload as a hypergraph, and utilizes that workload information to achieve a data placement that minimizes the number of distributed transactions [6].

3.2.1 System Architecture

The key components of SWORD are shown in Figure 3.1, and can be functionally divided into three groups: data partitioning and placement, incremental repartitioning, and transaction processing. The data itself is horizontally partitioned (sharded) across a collection of physical database partitions, i.e., machines that run a relational resource manager such as a relational DBMS server (PostgreSQL in our implementation). Data may be replicated (at the granularity of tuples) to improve availability, performance, and fault tolerance. We assume that each tuple is associated with a globally unique **primary key**, which may consist of more than one attribute. We briefly discuss the key functionality of the different components next.

Data partitioning and placement: These modules are in charge of making the initial workload-aware data placement and replication decisions, and then carrying out those decisions through appropriate data migration and replication. The query workload manager takes the query workload trace (the set of transactions and the tuples they access (Section 3.4.1)) as input and generates a *compressed* hypergraph representation (Sec-

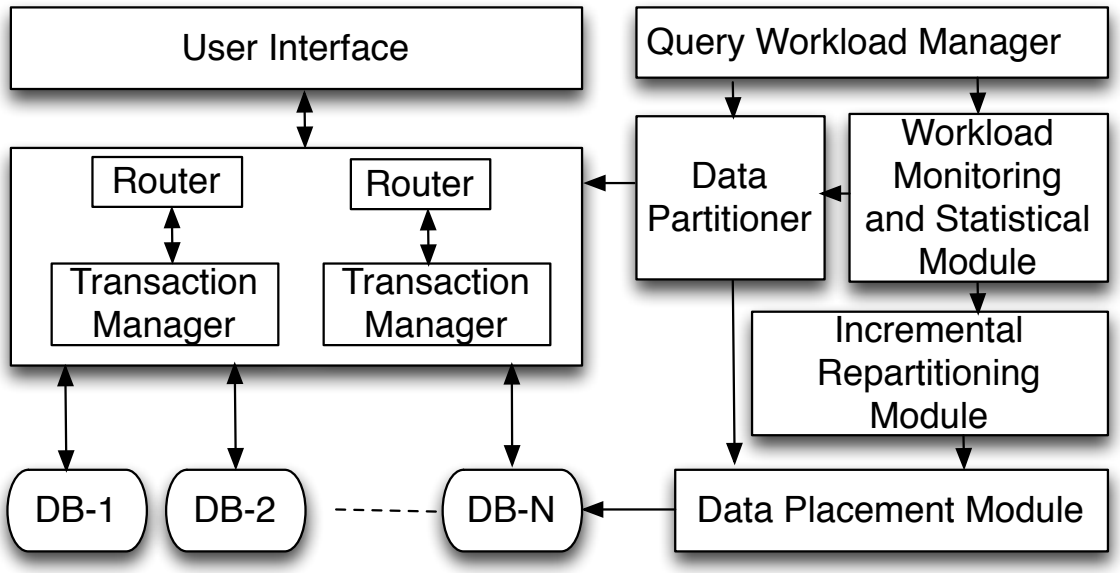


Figure 3.1: **System architecture**

tion 3.3.1) of the query workload. The compressed hypergraph is then fed to the data partitioner which does in-graph replication (Section 3.3.3), and partitions the resulting hypergraph using the hMetis [60] partitioning tool. The output of the partitioner is a mapping of the tuples to their physical database partitions. These mappings are fed to the data placement sub-module and the router. The data placement sub-module then uses these mappings to partition the database across the machines.

Incremental repartitioning: The workload monitoring and statistical module monitors the workload changes and maintains statistics on the workload access patterns. It provides this input to the incremental repartitioning module (Section 3.3.2) which identifies when the current partitioning is sub-optimal and triggers data migration to deal with workload changes. The data migration is done in incremental steps through the data placement module during periods of low activity.

Transaction processing: The users submit transactions, and receive their results through an interface provided by the transaction processing module. The user interface sends the transactions to the router which parses the SQL statements in the transactions using an SQL parser that we wrote. The router determines the tuples accessed by the transaction (more specifically, the primary keys of the tuples accessed by the transaction), their replicas, and their location information using the mappings provided by the data partitioner. The router also determines the appropriate number of replicas that need to be accessed for each tuple to satisfy the quorum requirements (Section 3.3.4). The router then uses a *set-cover based* algorithm to compute the minimum number of partitions that the transaction needs to be executed on (referred to as **query span** in the rest of the chapter), to access all the required tuples and replicas (Section 3.3.5). This information, along with the transaction, is passed on to the transaction manager which executes the transactions in parallel on the required database partitions. The transaction manager uses a *2-phase commit protocol* to provide the ACID guarantees.

We represent the query workload as a hypergraph, $\mathcal{H} = (V, E)$, where each hyperedge $e \in E$ represents a transaction, and the set of nodes $V_e \subseteq V$ spanned by the hyperedge represent the tuples accessed by the transaction. Each hyperedge is associated with an edge weight w_e which represents the frequency of such transactions in the workload.

A *k-way balanced min-cut partitioning* of this hypergraph would give us k balanced partitions of the database (i.e., k partitions of equal size) such that the number of transactions spanning multiple partitions is minimized. This is because every transaction that spans multiple partitions corresponds to a hyperedge that was cut in the partitioning.

Instead of looking for partitions of equal size, we may instead assign a *weight* to each node and ask for a partitioning such that the total weights of the partitions are identical (or almost identical). The weights may correspond to the item sizes (in case of heterogeneous data items) or some combination of the item sizes and access frequencies. The latter may be used to achieve balanced loads across the partitions.

The problem of k-way balanced min-cut partitioning generalizes the *graph bisection* problem, and is NP-hard. However, due to the practical importance of this problem, many efficient and effective hypergraph partitioning packages have been developed over the years. We use the hMetis package [60] in our implementation.

3.2.2 Workload Modeling

Figure 3.2(a) shows an illustrative example where a transactional query workload is transformed into a hypergraph. The hypergraph consists of a vertex set $V = \{a, b, c, e, f, g\}$ and hyperedge set $E = \{e_1 = \{a, b, c\}, e_2 = \{a, g\}, e_3 = \{g, c\}, e_4 = \{a, e\}, e_5 = \{f, c\}\}$ where e_i represent the transactions. A 2-way min-cut partitioning of this hypergraph gives us 2 distributed transactions, as compared to a naive round-robin partitioning that would have given us 4 distributed transactions.

3.3 System Design

In this section, we first present our proposed techniques for scalable workload-aware data partitioning, and for incremental repartitioning to cater to workload variations. We then discuss our in-graph replication mechanism, and use of fine-grained quorums to

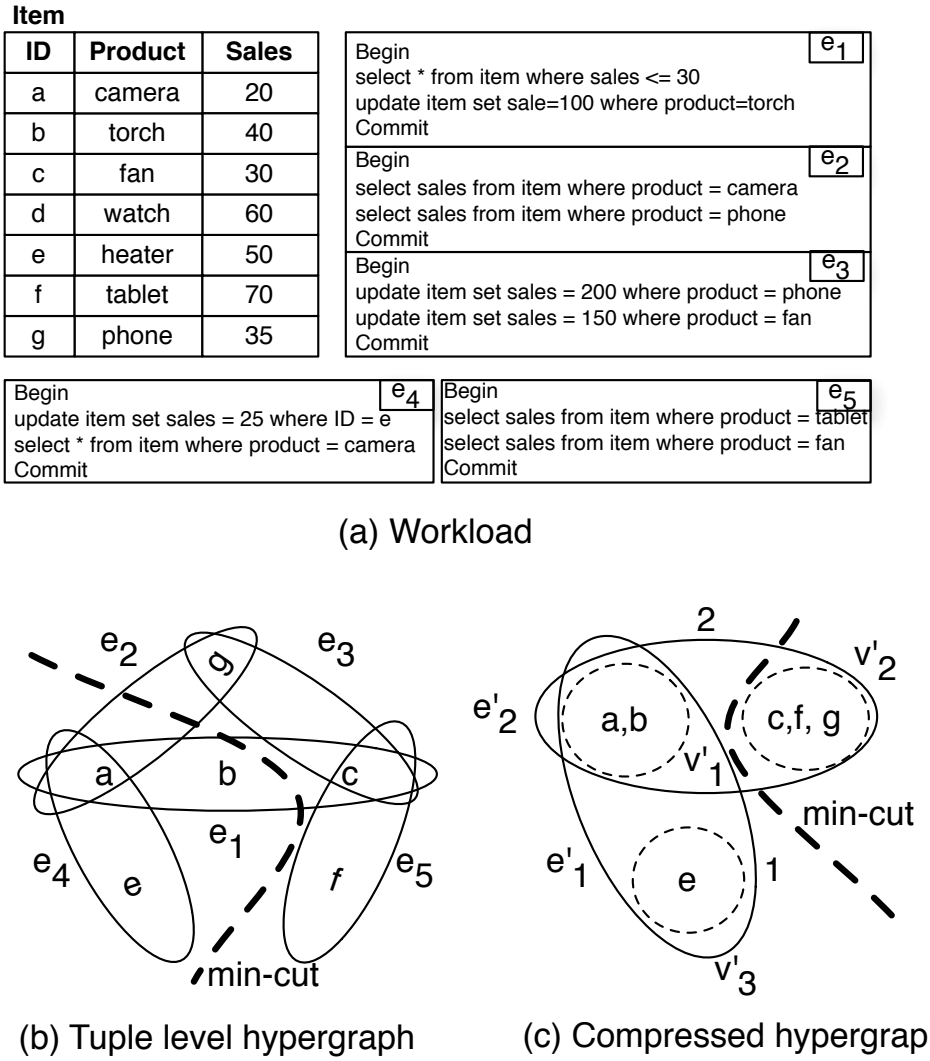


Figure 3.2: **(a) A sample workload; (b) Workload representation using a hypergraph; (c) Hypergraph compression.**

improve availability. Finally, we present our query routing mechanism to select partitions to involve in a given query or a transaction.

3.3.1 Hypergraph Compression for Scaling

The major scalability issues involved with workload-aware hypergraph (or graph) partitioning-based techniques are: (1) the memory and computational requirements of

hypergraph storage and processing, which directly impact the partitioning and repartitioning costs, and (2) the large size of the tuple-to-partition mapping produced by the partitioner that needs to be stored at the router for routing the queries to appropriate partitions, that makes the router itself a bottleneck in query processing. Existing hypergraph compression techniques [61] based on *coalescing* help in effectively reducing the size of the hypergraph, and in some cases [62] even minimize the loss of structural information by using additional neighborhood information as input to the coalescing function. However these techniques do not reduce the sizes of mapping tables required for routing queries, and thus are not appropriate for our context.

We propose using a simple two-step **hypergraph compression** technique instead. We first group the nodes of the hypergraph (i.e., database tuples) into a large number of groups using an easy-to-compute function applied to the primary keys of the tuples, and we then collapse each group of nodes into a single *virtual node*. More specifically, in the first step, we map each node $v \in V$ in the original hypergraph to a *virtual node* $v' \in V'$ in the compressed hypergraph by computing $v' = f(pk_v)$, where pk_v represents the node v's primary key. In our current implementation, we use a hash function, $HF(pk_v) = hash(pk_v) \bmod N$, where N is the desired number of virtual nodes. However, any inexpensive function (e.g., a range partitioner) could be used instead. Using such a primary key-based coalescing plays a crucial role in developing an efficient and scalable routing mechanism with minimum book-keeping; further details are discussed in Section 3.3.5.

Let V' denote the resulting set of virtual nodes. For a hyperedge $e \subseteq V$ in the original hypergraph, let $e' \subseteq V'$ denote the set of virtual nodes to which the vertices in e

were mapped. If e' contains at least two virtual nodes, then we add e' as a hyperedge to the compressed graph (denoted $\mathcal{H}' = (V', E')$). We define the hypergraph compression ratio (CR) as the ratio of the number of nodes $|V|$ in the original hypergraph to the number of virtual nodes $|V'|$ in the compressed hypergraph, i.e., $CR = \frac{|V|}{|V'|}$. $CR = 1$ indicates no compression, whereas $CR = |V|$ indicates that all the original vertices were mapped onto a single virtual node.

Next, we iterate over each hyperedge $e \in E$ of the original hypergraph and replace each node $v_e \in V_e$ spanned by the hyperedge e with $v'_{e'}$ using the mapping generated in the first step. Each hyperedge $e' \in E'$ so generated in the compressed hypergraph is associated with an edge weight $w_{e'}$ which represents the frequency of the hyperedge. Figure 3.2(b) provides an illustration of compressed hypergraph generation. The mappings produced by the first step create virtual nodes $v'_1 = \{a, b\}$, $v'_2 = \{c, f, g\}$ and $v'_3 = \{e\}$. The second step generates the hyperedges e'_1 and e'_2 and the edge weights associated with these hyperedges depict the frequency of transactions accessing the corresponding sets of virtual nodes.

This hybrid coarse-grained approach, although simple, is highly effective at reaping the benefits of workload-aware partitioning without incurring the high overhead of the fine-grained approach. First, the hypergraph size is reduced significantly, reducing the overhead of running the partitioning and repartitioning algorithms. Second, it naturally handles new inserted tuples and tuples that were not part of the provided workload. Each such tuple is assigned to a virtual node based on its primary key and placed on the partition assigned to the virtual node. Third, it avoids *over-fitting* the partitioning and replication to the provided workload, resulting in more robust data placement. We also

need significantly smaller query workloads as input to make partitioning decisions.

On the other hand, the coarseness introduced by the compression process may result in larger min-cuts (and thus higher number of distributed transactions). However, we empirically show in Section 3.4 that the orders-of-magnitude gains in terms of the above mentioned benefits far offset the probable increased cost of distributed transactions as compared to a fine-grained approach.

3.3.2 Incremental Repartitioning

A workload-driven approach is susceptible to performance degradation if the actual workload (in the future) is significantly different from the workload used to make the partitioning and replication decisions. The quantum of performance variance is dependent on the sensitivity of the data placement technique to workload change. As we illustrate through our experimental evaluation (Section 3.4.4), the coarser representation achieved through hypergraph compression makes our approach less sensitive to workload changes compared to the fine-grained approach. However, significant workload changes will result in the initial placement being sub-optimal over time. In this section, we present an incremental repartitioning technique that performs data migration in incremental steps without resorting to complete repartitioning.

Our proposed incremental repartitioning technique monitors the workload changes at regular intervals, and moves a fixed amount of data items across partitions in incremental steps to mitigate the impact of workload change. The data migration is triggered whenever the percentage increase in the number of distributed transactions ($\triangle_{mincut}$) crosses

a certain *threshold*, c, a system parameter which can be set as a percentage of the initial min-cut, depending upon the sensitivity of applications to latency.

At a high level, our algorithm maintains pairs of sets of candidate virtual nodes that can be *swapped* to reduce the size of the min-cut. During lean periods of activity, the algorithm makes a maximum of k such moves in each step to reduce the min-cut of the data placement as per the current workload. It repeats these steps until the min-cut reduces below the threshold value. The algorithm thus provides an incremental approach to adjust the data placement without resorting to complete data migration.

More specifically, let $\mathcal{H}_{cut} = \{e_1, e_2, \ldots, e_t\}$ denote the set of hyperedges that span multiple partitions, i.e., the set of hyperedges in the cut, as per the initial data placement. Let $\mathcal{P}_{cut} = \{P_1, P_2, \ldots, P_t\}$ be the set of *partition sets*, where $P_i \in \mathcal{P}_{cut}$ is the set of partitions spanned by hyperedge $e_i \in \mathcal{H}_{cut}$. Further, let $V_i = \{v_1, v_2, \ldots, v_n\}$ be the set of virtual nodes covered by hyperedge e_i, and let $\mathcal{V}_{cut} = \{\bigcup_{i=1,\ldots,t} V_i\}$, be the union set of nodes covered by all the hyperedges in the cut. This is the first set of our *candidate nodes* for migration. For each virtual node $v_i \in \mathcal{V}_{cut}$, in our first candidate set, we maintain a set of partitions $\mathcal{P}_{v_i}$ that contain the node or its replicas, such that $\{v_i \in p_j, \forall p_j \in \mathcal{P}_{v_i}\}$. Let nh_{ij} be the sum of the weights of hyperedges incident on node v_i in partition p_j. So each vertex v_i is associated with a set $\mathcal{NH}_i$ where $nh_{ij} \in \mathcal{NH}_i$ and $p_j \in \mathcal{P}_{v_i}$.

Let $\mathcal{VS}$ be the set of virtual nodes that are covered only by hyperedges that are not cut. In other words, if we move a virtual node in $\mathcal{VS}$ to a different partition, it would not change the min-cut value. This set of virtual nodes forms our second set of candidate nodes.

Let the contribution of each hyperedge $e \in \mathcal{H}_{cut}$ towards total number of distributed

transactions seen so far be

$$C_e = \frac{ndt_e}{\sum_{i=1,\ldots,t} ndt_{e_i}}$$

and let $\mathcal{C} = \{C_e | e \in H_{cut}\}$. The numerator ndt_e is the weight of the hyperedge e in the cut, whereas the denominator is the sum of the weights of the hyperedges in the cut. We maintain a priority queue, $\mathcal{PQ}$ of hyperedges $e \in \mathcal{H}_{cut}$. Each element in $\mathcal{PQ}$ is ordered by C_e and thus the largest element represents the hyperedge with the highest value of C_e. We choose to consider only those hyperedges that span two partitions which guarantees that a single swap of virtual nodes between two partitions would reduce the min-cut[3].

Swapping gain ($\mathcal{SG}$): Consider a hyperedge $e_i \in \mathcal{H}_{cut}$ spanning two partitions $p_a \in P_i$ and $p_b \in P_i$ where $P_i \in \mathcal{P}_{cut}$. Let $S_a = \{p_a \bigcap V_i\}$, $S_b = \{p_b \bigcap V_i\}$, be the set of virtual nodes covered by e_i in the partitions p_a and p_b respectively. Let $\bar{S_a} = \{\{p_a - V_i\} \bigcap \mathcal{VS}\}$, $\bar{S_b} = \{\{p_b - V_i\} \bigcap \mathcal{VS}\}$. The swapping of all the virtual nodes in S_b with a set of virtual nodes $\{I \subseteq \bar{S_a} \mid I_w \simeq S_{b_w}\}$ where I_w and S_{b_w} is the sum of node weights in I and S_b respectively (to maintain a load balance), would result in two things. Firstly e_i would be removed from $\mathcal{H}_{cut}$ decreasing the min-cut by ndt_e. Secondly, the set of hyperedges other than e_i which are incident on the nodes in S_b might probably become distributed increasing the min-cut by $(\sum_{i \in S_b} nh_{ib} - ndt_e)$ in the worst case. Thus the minimum swapping gain $\mathcal{SG}$ is given by:

$$\mathcal{SG} = ndt_e - (\sum_{i \in S_b} nh_{ib} - ndt_e) = 2 \times ndt_e - \sum_{i \in S_b} nh_{ib}$$

[3] A majority of hyperedges in the cut of our compressed hypergraph representing TPC-C, a typical OLTP workload, span two partitions.

Algorithm 1: Incremental repartitioning algorithm

Require: Initial min-cut M_c, $\mathcal{PQ}$, threshold c, $CN = \emptyset$.

1: **while** $\triangle_{mincut} > c\%$ of M_c **do**
2: **while** $|CN| < k$ **do**
3: $e = \mathcal{PQ}.peek()$
4: $\mathcal{SG}_1 = 2 \times ndt_e - \sum_{i \in S_a} nh_{ia}$
5: $\mathcal{SG}_2 = 2 \times ndt_e - \sum_{i \in S_b} nh_{ib}$
6: **if** $\mathcal{SG}_1 \geq \mathcal{SG}_2$ and $\mathcal{SG}_1 > 0$ **then**
7: Identify $\{I \subseteq S_b \mid I_w \simeq S_{a_w}\}$
8: $CN = CN \leftarrow (I, S_a)$
9: $\mathcal{PQ}.remove(e)$
10: **else if** $\mathcal{SG}_2 \geq \mathcal{SG}_1$ and $\mathcal{SG}_2 > 0$ **then**
11: Identify $\{I \subseteq S_a \mid I_w \simeq S_{b_w}\}$
12: $CN = CN \leftarrow (I, S_b)$
13: $\mathcal{PQ}.remove(e)$
14: **end if**
15: **end while**
16: Swap the k sets of virtual nodes
17: Update $\mathcal{H}_{cut}$, $\triangle_{mincut}$, $\mathcal{PQ}$
18: **end while**

Algorithm 1 provides the details of our proposed incremental partitioning technique. A background process monitors the workload and populates $\mathcal{PQ}$. The algorithm is triggered when $\triangle_{mincut}$ exceeds a given threshold value c. The algorithm (lines 2-9) identifies at most k pairs of sets of virtual nodes for swapping which would maximize the total SG and stores them in $\mathcal{CN}$ as candidates to be swapped. It executes the k swaps (line 10) at a lean period of activity. It then updates $\mathcal{H}_{cut}$, $\triangle_{mincut}$ to reflect the changes caused by the swaps. It repeats the steps until the current min-cut falls below the set threshold value.

3.3.3 Workload-aware Replication

Active and aggressive replication has the potential to provide better load balancing, improved availability in presence of failures, and a reduction in the number of distributed *read* transactions. However, providing strict transactional semantics with ACID properties becomes a challenge in presence of active replication [33].

We propose an aggressive workload-aware replication technique that provides data availability proportional to the workload requirement. We exploit tuple-level access pattern statistics to ascertain the number of replicas for each data item. We argue that the drawbacks of replicating items that are heavily updated are offset by several considerations: (1) for availability, it is desired that each data item be replicated at least once; (2) items that are heavily updated are typically also heavily read and replicating those items can reduce the total number of read-only distributed transactions; (3) through use of appropriate *quorums*, we can balance the writes across a larger number of partitions. A key feature of our replication technique is the notion of disentangling transactions to afford better min-cuts. We discuss this further below.

Replica generation: We have developed a *statistical module* that uses the transactional logs (Section 3.4.1) to compute the read and write statistics for each virtual node. Each node v'_i in the compressed hypergraph $\mathcal{H}$ represents a set of tuples T_i. Each tuple $t_{ij} \in T_i$ has a read frequency (fr_{ij}) and a write frequency(fw_{ij}). To compute each node's replication factor we compute the size compensated average of reads and writes per virtual

node as follows:

$$Avg(v'_i)_w = \frac{\sum_j fw_{ij}}{logS(v'_i)},\ Avg(v'_i)_r = \frac{\sum_j fr_{ij}}{logS(v'_i)},\ \mathcal{R} = \frac{Avg(v'_i)_w}{Avg(v'_i)_r}$$

where $Avg(v'_i)_w$ and $Avg(v'_i)_r$ are average read and write frequencies of node v'_i, $logS(v'_i)$ is the log of the size of node v'_i. $\mathcal{R}$ is the average write-to-read ratio.

Based on the access pattern statistics generated, if the ratio $\dot{\mathcal{R}} \geq \delta$ (where $0 < \delta < 1$) the virtual node is replicated only once to control the cost of distributed updates. For all other nodes, the number of replicas generated is a linear function of $Avg_r(i)$. δ serves as a threshold value for controlling the number of replicas for heavily written tuples and can be chosen based on the workload requirements and the level of fault tolerance required. We use the log of the sizes of virtual nodes to compensate for the *skew* in the size of the virtual nodes; this helps in limiting the number of replicas created for heavily accessed large virtual nodes.

In-graph replication: Once we have chosen the number of replicas for a virtual node, we modify the compressed hypergraph by adding as many copies of the virtual node as required. One key issue then is assigning these virtual node replicas to the hyperedges in the graph. We observe that by doing this cleverly, we can disentangle some of the transactions that share data items and construct a graph with a better min-cut. Let $R_{v'}$ be the set of replicas for the virtual node v'. The replica assignment algorithm computes a set of distinct hyperedges $E_{v'}$ incident on v' and for each $e' \in E_{v'}$ its associated edge weight $w_{e'}$. There are two possibilities:

Case 1: $|E_{v'}| \geq |R_{v'}|$: We reduce this case to a simple *multi-processor scheduling* prob-

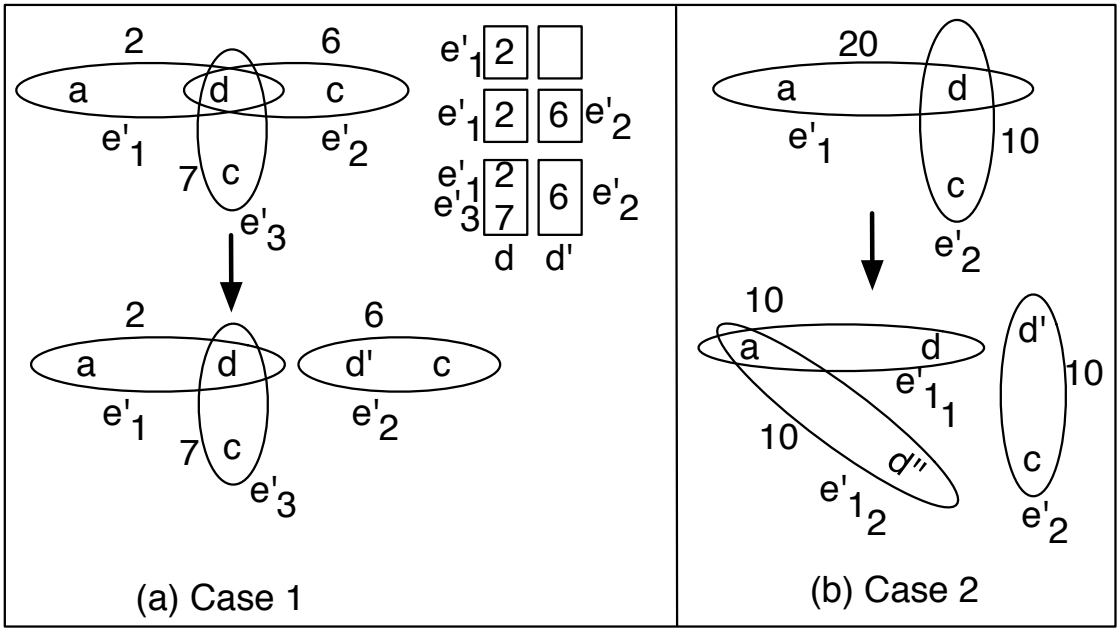


Figure 3.3: **In-graph replication**

lem. Each replica $r \in R_{v'}$ is associated with a processor b_r. Each hyperedge $e' \in E_{v'}$ is assigned to one of these processors, increasing the current load on the processor by the corresponding hyperedge weight $w_{e'}$. Minimizing the maximum load across the processors is equivalent to finding an equitable assignment of the replicas to the hyperedges. Since the scheduling problem is NP-Hard, we use a greedy approach which considers the hyperedges in the decreasing order by weight, and assigns the next hyperedge to the processor with the current minimum load. Finally all the hyperedges assigned to a particular processor b_r are allocated the replica r associated with the processor.

Figure 3.3(a) gives an example showing the assignment of a virtual node d and its replica d' to the incident hyperedges e'_1, e'_2 and e'_3 with edge weights 2, 6 and 7 respectively. The algorithm creates two processors representing d and its replica $d^{'}$ and assigns hyperedges to these processors greedily.

Case 2: $|E_{v'}| < |R_{v'}|$: The insertion algorithm splits the hyperedge e' with the highest weight $w_{e'_{max}}$ into two hyperedges with weights $\lceil \frac{w_{e'_{max}}}{2} \rceil$, $\lfloor \frac{w_{e'_{max}}}{2} \rfloor$ respectively. It repeats this procedure until the number of hyperedges is equal to $|R_{v'}|$ and then allocates one replica to each hyperedge.

Figure 3.3(b) gives an example for case 2 showing the assignment of a virtual node d and its replicas d', d'' to the incident hyperedges e'_1 and e'_2 with edge weights $w_{e'_1} = 20$ and $w_{e'_1} = 10$ respectively. The algorithm chooses e'_1 since it has the highest weight and splits it into two hyperedges e'_{1_1} and e'_{1_2} each with a weight of 10.

3.3.4 Fine-grained Quorums

Aggressive active replication comes at the cost of distributed update transactions which hurt performance. *Quorums* [63] have been extensively used to control the overheads associated with distributed updates for maintaining active replica consistency [33, 34]. In addition to this, quorums also help in improving fault tolerance by gracefully dealing with partition failures.

Let $S = \{S_1, S_2, \cdots, \}$ denote the set of partitions on which a data item is stored. A quorum system Q (for that data item) is defined to be a set of subsets of S with pair-wise non-empty intersections [32]. Each element of Q is called a quorum. A simple example of a quorum system is the *Majority quorum*, where every majority of the partitions forms a quorum. Defining read and write quorums separately, a quorum system is valid if: (a) every read quorum (rq) overlaps with every write quorum (wq), and (b) every two write quorums have an overlap. Another quorum system is ROWA (read-one-write-all), where

a read can go to any of the partitions, but a write must go to all the partitions. Quorums allow us to systematically reduce the number of partitions that must be involved in a query, without compromising correctness.

Depending on the nature of the workload, the choice of the quorum system plays a significant role in determining its effectiveness in improving performance. For example, ROWA quorum would perform well for read intensive workloads and Majority quorum would help in controlling the cost of distributed updates for write intensive workloads. However different transactional workloads might have different mixes of read and write queries. Also, different data items in a given workload may have different read-write access patterns. Choosing a *fixed quorum for the all the data items* in the system a priori may significantly hurt the performance.

In this dissertation, we propose using *fine-grained quorums*, which are defined at the virtual node level (a group of tuples). We focus on two quorum systems, ROWA and Majority. Given a workload, the type of quorum for each virtual node is decided based on its read/write access pattern, as monitored by the statistical module. We compute $\mathcal{R}$, the write-to-read ratio (Section 3.3.3) for each virtual node. The quorum for each virtual node is then decided based on the value of $\mathcal{R}$. If $\mathcal{R} > \gamma$, where $(0 < \gamma < 1)$, then we chooses Majority quorum else ROWA quorum. The value of γ is a system parameter, which can be adjusted based on the nature of the query workload. We experimented with different values for γ and observed that as γ increases from 0.5 and tends towards 1, the system chooses ROWA for most data items incurring a high penalty for writes thereby reducing performance. On the other hand, as γ decreases from 0.5 and tends towards 0, the system chooses Majority quorum for most data items incurring a higher overhead for

reads. Our experiments showed that $\gamma = 0.5$ was able to achieve a fine balance between the benefits of ROWA quorum for reads and Majority quorum for reducing the number of copies to be updated and gave the best performance for the TPC-C benchmark.

Quorums defined at the virtual node level specify the number of copies of each data item that need to be accessed in order to meet the quorum requirement. For each virtual node v' having a set of available copies $\mathcal{C}_{v'}$, a read quorum $|c_r|$, $c_r \subset \mathcal{C}_{v'}$ and a write quorum $|c_w|$, $c_w \subset \mathcal{C}_{v'}$ defines the number of copies of v' required for either a read or write query. These read and write quorums values are defined based on the types of quorum. For example, a majority quorum requires that $|c_r| + |c_w| > |C_{v'}|$ and $2 * |c_w| > |C_{v'}|$, while ROWA requires $|c_r| = 1$ and $|c_w| = |C_{v'}|$.

The choice of quorum at the level of each virtual node makes the system adaptive to a given workload and improves the effectiveness of quorums in reducing the costs of distributed updates significantly. We have conducted extensive experiments to study the use of different quorums for a number of query workloads with different mixes of read and writes. Our results show that fine-grained quorums provide significant benefits in terms of reducing the average query span and improving the transaction throughput for different types of workloads. This feature is especially useful for database-as-a-service in a cloud computing environment.

3.3.5 Query Routing

The use of graph based partitioning and replication schemes requires that the mappings of tuples to partitions be stored at the router to direct transactions to appropriate

partitions. This is a major scalability challenge since the size of these mappings can become very large, and they may not fit fully in the main memory leading to increased lookup times. The problem is further aggravated with tuple-level replication which only adds to the size of these mappings. Existing techniques for dealing with this issue [23] use compute-intensive look-up table compression techniques coupled with a scaled-up router architecture to fit the lookup tables in memory, which may not be cost effective.

We propose a routing mechanism that requires minimum book-keeping as a natural consequence of our hypergraph compression technique. The size of the mapping tables is reduced by a factor of CR (the hypergraph compression ratio). Depending on the router's compute and memory capacity, a suitable CR could be chosen to optimize overall performance. In addition to this, we incorporate two additional features to reduce the query span and the cost of distributed updates: fine-grained quorums (as described in Section 3.3.4) that determine the number of copies of each data item required, and a set-cover algorithm that determines the minimum number of partitions required to satisfy the query and meet the quorum requirements.

Minimum set-cover algorithm: The minimum set-cover problem to minimize the query span can be defined as follows: given a transaction e', a set of virtual nodes $V'_{e'}$ accessed by e' and their replicas $R_{e'}$; a set of partitions $\{P^{e'}_{RV'} \mid V'_{e'} \cup R_{e'} \subseteq P^{e'}_{RV'}\}$; a universe $\mathcal{U}_{e'} = \{v' \rightarrow c \mid v' \in V'_{e'}, c \in C_{e'}\}$ where c is the number of copies required for v' as per the quorum requirement; a set-cover map $\mathcal{S}_{e'} = \{v' \rightarrow c \mid v' \in V'_{e'}, c \in C_{e'}\}$ where the initial count c of each element is set to 0; determine the minimum number of partitions $S \subseteq P^{e'}_{RV'}$ that cover the universe $\mathcal{U}_{e'}$. The minimum set-cover is an NP-

Algorithm 2: Set-cover Algorithm

Require: $\mathcal{H}', e' \in E', P_i \in P_{RV}, \mathcal{U}_{e'} = \{v' \rightarrow c \mid v' \in V_{e'}, c \in C_{e'}\}, S_{e'} = \{v' \rightarrow c \mid v' \in V_{e'}, c \in C_{e'}\}$

1: **while** $UC_{e'} \neq 0$ **do**
2: p_{index} =argmax$_i(\{\mathcal{U}_{e'} - S_{e'}\} \cap P_i)$
3: $S \cup = P_{p_{index}}$
4: $S_{e'} += P_{p_{index}}$
5: $UC_{e'} = \mathcal{U}_{e'} - S_{e'}$
6: **end while**
7: **return** S

Complete problem and we use a greedy heuristic to solve the same. In each iteration the algorithm determines the partition P_i which covers the maximum uncovered elements $UC_{e'}$ in the universe $\mathcal{U}_{e'}$ given by $max(\{\mathcal{U}_{e'} - S_{e'}\} \cap P_i)$, where $\{\mathcal{U}_{e'} - S_{e'}\}$ denotes the operation wherein the counts of the elements in the universe $\mathcal{U}_{e'}$ are decremented by the count of the corresponding elements in $S_{e'}$. The set-cover S is updated with the partition P_i, i.e., $S = S \cup P_i$, $S_{e'} = S_{e'} + P_i$ which increases the count of common elements in the set-cover map by one. The uncovered elements are updated by $UC_{e'} = \mathcal{U}_{e'} - S_{e'}$ which reduces the counts of common elements in $\mathcal{U}_{e'}$ by the counts of the corresponding elements in $S_{e'}$. The algorithm terminates when the counts of all elements in $UC_{e'} = 0$ and outputs S. The algorithm for computing the set-cover is shown in Algorithm 2.

To give an example: consider a transaction $e' = \{2, 3, 5, 9, 12, 14\}$ where the numbers in the set indicate the IDs of the virtual nodes accessed by e', $C_{e'} = \{2, 1, 1, 2, 1, 1\}$ which denotes the number of copies required for corresponding elements in e' to satisfy the quorum requirements. Consider a set of partitions $P_1 = \{2, 9\}$, $P_2 = \{2, 3^c, 5^c, 14\}$, $P_3 = \{9^c, 5^c, 12\}$, and $P_4 = \{12^c, 14^c\}$ where the element $n \in V'_{e'}$ and $n^c \in R_{V'_{e'}}$. In the first iteration the algorithm chooses P_2, updates $S = \{P_2\}$, then computes the uncov-

Table 3.1: **Router memory requirements**

Scheme	Fine-grained	CR=3	CR=6	CR=11	CR=28
Mappings size	20 GB	8GB	4GB	2.2GB	857MB

ered elements $UC_{e'} = e' - P_2$ updating $C_{e'} = \{1, 0, 0, 2, 1, 0\}$. In the second iteration it chooses P_3, updates $S = \{P_2, P_3\}$ and $C_{e'} = \{1, 0, 0, 1, 0, 0\}$. In the third iteration the algorithm chooses P_1, updates $S = \{P_2, P_3, P_1\}$ and $C_{e'} = \{0, 0, 0, 0, 0, 0\}$ and the algorithm terminates as all elements in the universe are covered. S constitutes the minimum number of partitions that the transaction needs to be routed to.

Generation of mapping tables: We use a hash table-based lookup mechanism to determine the virtual nodes accessed by a query. We create a set of hashmaps for each relation $\mathcal{RE}_i$ in the partitioned database. Using the hash function HF_{pk_v} used for graph compression to map tuples to virtual nodes, for each relation $\mathcal{RE}_i$, we create one hashmap $\mathcal{M}_k$ per primary key attribute $k \in \mathcal{K}_i$ where $\mathcal{K}_i$ is the set of attributes which form the primary key for relation $\mathcal{RE}_i$. These hash tables map the distinct values of $k \in \mathcal{K}_i$ to a set of virtual nodes that contain tuples with corresponding values of k. Further, we create union maps $\mathcal{M}_i = \cup \mathcal{M}_k, \forall k \in \mathcal{K}_i$ which essentially contain all virtual nodes containing tuples of relation $\mathcal{RE}_i$. These mapping tables generated at the virtual node level need to be updated or regenerated only at the time of data repartitioning, a relatively infrequent process for stable workloads. New tuples in the database are automatically mapped to existing virtual nodes and hence do not require any updates to the mapping tables at the router.

We see a drastic reduction in memory requirement compared to the fine-grained scheme. This can be attributed to two factors. First, the tables are maintained at the level of virtual nodes and hence provide a reduction in size by a factor of CR. Second, we

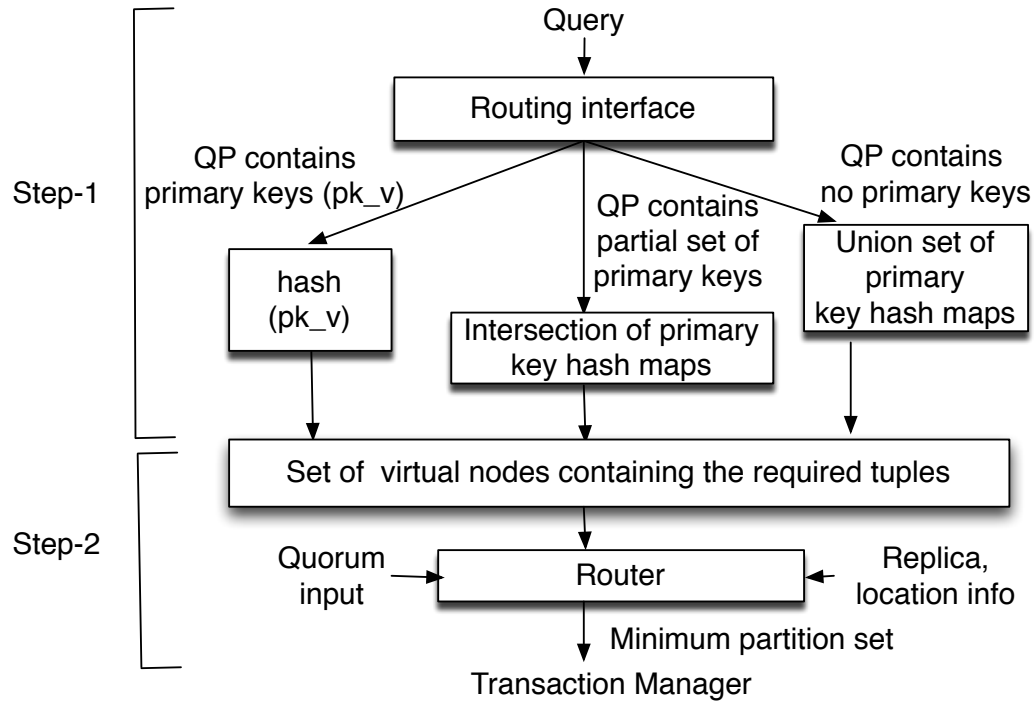


Figure 3.4: **Routing architecture**

maintain hash maps per primary key attribute. The number of distinct values per primary key attribute is much smaller than the total number of distinct primary key values[4], making the hash tables very compact. Table 3.1 shows the effectiveness of our proposed routing mechanism by comparing the size of the router mapping tables for a workload of 1 Billion tuples.

Routing mechanism: Figure 3.4 illustrates the flow of our routing mechanism. The routing interface provided by the query processing module takes a query as input and parses it to determine the relation $\mathcal{R}_i$ and the set of primary keys $\mathcal{QP}_i$ in the query predicate. It then deals with three cases. First, if $\mathcal{QP}_i = \mathcal{K}_i$, it simply hashes the key values and obtains the virtual node which the query needs to access. This process requires no lookup and is the

[4]The Cartesian product of the full set of attributes forming the primary key.

most efficient case. Second, if $\mathcal{QP}_i \subset \mathcal{K}_i$ the routing module returns $\cap \mathcal{M}_k, \forall k \in \mathcal{QP}_i$ which would give the set of all virtual nodes that contain tuples with the corresponding primary key values. The lookup and intersection operations are quite efficient as the size of these tables is small and the operations can be done in memory. Third, if $\mathcal{QP}_i = null$, i.e., the query predicates do not contain any primary key attributes, the union set $\mathcal{M}_i$ of the corresponding relation $\mathcal{R}_i$ is returned. Here no computation is involved as the pre-computed union set is returned.

In the next step, the router determines the virtual node replicas, and their location information using the mappings obtained from the data partitioner. It gives this information as input to the set-cover algorithm which computes the minimum partition set that meets the quorum requirement on which the transaction needs to be executed.

3.4 Experimental Evaluation

In this section, we present the results of the experimental evaluation of our system. We first provide some details of our system implementation in Section 3.4.1 followed by our experimental setup in Section 3.4.2. We then provide an experimental analysis of our hypergraph compression technique, and discuss the effects of our techniques on router efficiency and system throughput. We then evaluate our fine-grained quorum technique and its effect on the end-to-end system performance.

3.4.1 System Implementation

We have used PostgreSQL 8.4.8 as the relational DBMS system and Java-6-OpenJDK SE platform for developing and testing our framework. For hypergraph generation we follow an approach similar to that of *Schism* [6] wherein we use the PostgreSQL logs to determine the queries run by the benchmark. We have developed a query transformation module that transforms each query into an equivalent SELECT SQL query, from which we can extract the primary keys of the tuples accessed by the query to build the hypergraph. For executing the transactions the transaction manager uses the Java transaction API's (JTA) XAResource interface to interact with the DBMS resource managers running on the individual partitions and executes transactions as per the 2-phase commit protocol.

3.4.2 Experimental Setup

This section provides the details of our system configuration, workloads, datasets and the baselines used.

3.4.2.1 System Configuration

Our system deployment on Amazon EC2 consists of one router cum transaction manager and 10 database partitions each running an instance of a PostgreSQL 8.4.8 server running with a *read committed* isolation level. The router configuration consists of 7.5 GB memory, 4 EC2 Compute Units, 850 GB storage, and a 64-bit platform with Fedora Core-8. The 10 database partitions are run on separate EC2 instances each with a configuration of 1.7 GB memory, 1 EC2 compute unit, 160 GB instance storage, 32-bit platform with

Fedora Core-8.

3.4.2.2 Workloads and Datasets

We have used the TPC-C benchmark for our experimental evaluation which contains a variety of queries: 48% write queries (update and insert), 47% read queries (select), and 5% aggregate queries (sum and count). The database was horizontally partitioned into ten partitions according to different partitioning schemes for the purposes of experimental evaluation. In order to experiment with a variety of different configurations, we used a dataset containing 1.5 Billion tuples, and different transactional workloads consisting of up to 10 million transactions. We have varied the percentage of read and write transactions to simulate different types of transactional workloads. In particular, we created three different mixes from the TPC-C workload: *Mix-1* consisting of 75% of read-only transactions and 25% write transactions, *Mix-2* consisting of 50% each of read and write transactions, and *Mix-3* consisting of 25% read and 75% write transactions.

3.4.2.3 Baselines

In our experiments we compare the performance of our approach of using compressed hypergraphs (referred to as *compressed*) with two partitioning strategies as baselines: *Random (hash-based)* with 3-way replication, and *fine-grained* tuple-level hypergraph partitioning approach.

Random: We use tuple-level random partitioning with 3-way replication as our baseline. This approach is essentially the same as *hash* partitioning. We place the tuples using a

hash function (specifically, by overriding the hashcode() function in java to a hash function of choice, MD5 in our case) with range $\{1, ..., P\}$ (P = number of partitions), and the resulting placement is nearly random. The 3-way replication is achieved using 3 different hash functions, with a post-processing step to ensure no two replicas land on the same partition.

We note here that the TPC-C benchmark includes a large number of queries that access a small number of tuples, and a few queries that access a large number of tuples. In particular, we need to be able to handle queries where only **a portion of the primary key of a table is specified**. An example of such a query is:

select count (c_id) from customer where c_d_id = 3

and c_last = 'OUGTHCALLYPRES' and c_w_id =1.

The predicates of this query specify a partial primary key set and the query accesses more than one tuple. Therefore, in spite of using hash functions to map the tuples and their replicas to physical partitions, we still need tuple-level mapping tables to determine the locations of all tuples that are accessed by such queries.

Fine-grained: Fine-grained partitioning is obtained by first constructing a tuple-level hypergraph where each hyperedge represents a transaction and nodes spanned by the hyperedge represent tuples accessed by the transaction. We use tuple-level read-write access patterns to determine the number to replicas for each tuple and use in-graph replication which approximates an average 3-way replication for each tuple, and partition the hypergraph using hMetis to obtain a fine-grained partitioning.

Compressed: We generated compressed hypergraphs for different compression ratios and

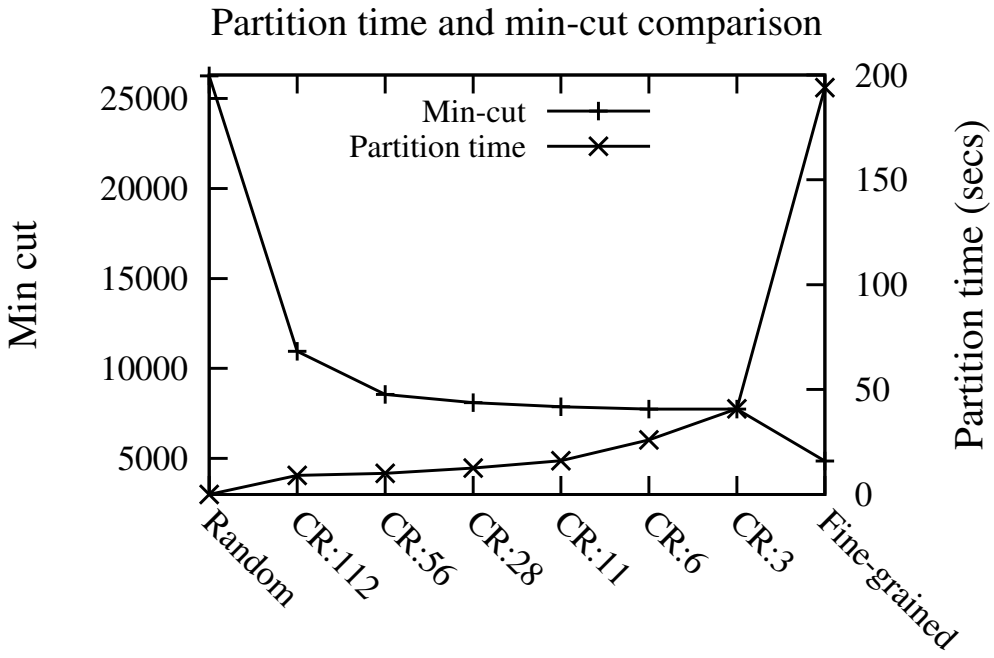


Figure 3.5: **Effect of hypergraph compression on min-cut and partitioning time. (Note that the left y-axis does not start at 0.)**

selected six different CRs (3, 6, 11, 28, 56, 112) as candidates for comparing our proposed technique with random and fine-grained partitioning schemes for our initial experiments in Section 3.4.3. Based on the results obtained we use a subset of these CRs (6, 11 and 28) for our experiments in the subsequent sections. Although our workload-aware replication scheme generates different number of replicas for each virtual node based on its access patterns, our scheme approximates an average 3-way replication in terms of the total number of replicas produced to provide a fair comparison with the other two techniques.

3.4.3 Hypergraph Compression Analysis

We explored the trade-off between the partitioning quality (min-cut) and the partitioning time for different compression ratios (CRs) and compared the same with our baselines (Figure 3.5). The number of distributed transactions is highest for the random

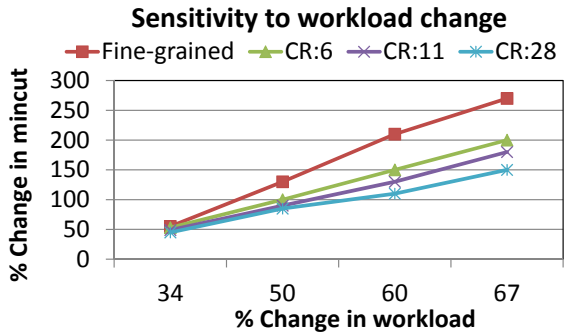
partitioning scheme (26244) since it does not take into account the nature of the query workload at all. The min-cut for fine-grained is the minimum (4860) as it accurately represents the query workload at the tuple-level. The min-cuts for the compressed graphs lie in between random and fine-grained, and their magnitudes vary closely in accordance with the compression ratio of the hypergraph, ranging from 10944 for a compression ratio of 112, to 7740 for a compression ratio of 3. On the other hand, the hypergraph partitioning time is highest for the fine-grained and decreases significantly with the increase in the CR of the hypergraph. The partitioning time for random is 0 since it does not involve hypergraph partitioning and places the tuples randomly on different partitions.

There is a clear trade-off between the partitioning time and the min-cut. On one hand, a decrease in min-cut represents a reduction in the number of distributed transactions while a reduction in the partitioning time plays a crucial role in reducing the overall costs associated with partitioning and repartitioning the database. An interesting thing to note here is that there is little variation in the min-cut as the compression ratio is increased from 3 to 56 which gives us the flexibility of compressing the graph without paying too much penalty in terms of the increase in the number of distributed transactions and at the same time making the system more scalable and efficient in terms of handling larger query workloads. Based on these results we have chosen CRs 6, 11, 28, as potential sweet spots which have a reasonable min-cut and a substantially lower partitioning time for our further experiments. We advocate using an analogous analysis phase to choose the CR for other scenarios.

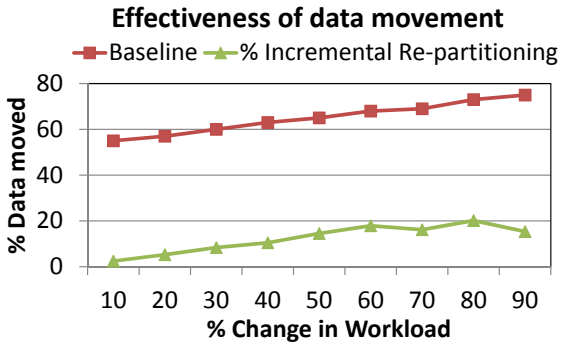
3.4.4 Effect of Workload Change

To ascertain the sensitivity of our approach to workload change and its impact on system performance, we conducted an experiment (Figure 3.6(a)) to evaluate the percentage change in the number of distributed transactions against the variation in the workload. For the purpose of the experiment, data was partitioned as per different partitioning strategies (fine-grained, our compressed hypergraph scheme with CRs 6, 11 and 28) for a given workload. Thereafter, the workload was varied by removing some old transactions and adding some new transactions. The variation in the number of distributed transactions was observed against the percentage change in workload. As can be seen, fine-grained partitioning was the most sensitive to workload change and the compressed hypergraph schemes were able to absorb the effect of workload change to a much greater extent with a smaller change in the number of distributed transactions for the same percentage change in workload.

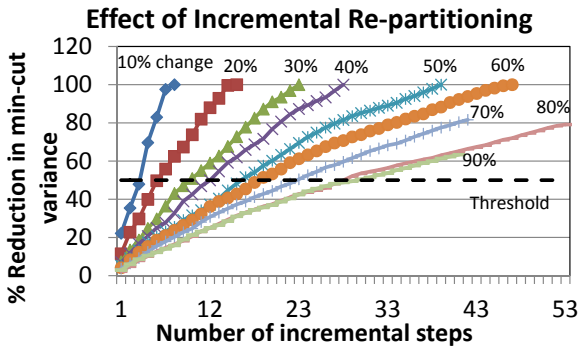
Experimental evaluation of our incremental partitioning module highlights its effectiveness in dealing with the performance variation due to workload change. Figure 3.6(b) shows the number of data items that need to be moved in terms of percentage of the total number of data items placed as the workload changes. We see that our scheme can handle up to a 90% change in workload by migrating up to a maximum of 20% of data items as compared to the baseline which represents the amount of data required to be migrated when performing a complete repartitioning of the database. Figure 3.6(c) shows the number of incremental steps required to bring the increased min-cut (due to workload change) to a value below the required threshold value.



(a)



(b)



(c)

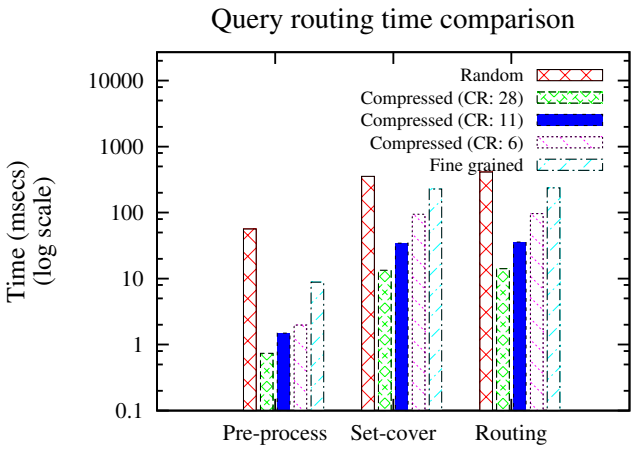
Figure 3.6: **(a) Fine-grained approach is more sensitive to workload changes than the compressed approach; (b) Our approach needs to move significantly smaller amount of data to maintain an effective partitioning compared to a baseline that does complete repartitioning; (c) Number of iterations required to bring down the increased min-cut under the threshold value.**

We see that the number of iterations required to bring the variation (or increase) in min-cut below the threshold value increases as the % change in workload increases. We compute the % change of workload in terms of the fraction of hyperedges (transactions) affected by the workload change. The number of steps would vary with the value of k, the number of swaps that can be done in one iteration. The results shown are for $k = 10$.

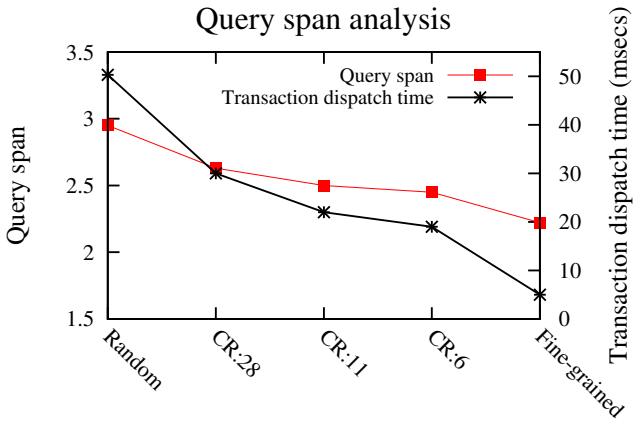
3.4.5 Routing Efficiency and Quality

We measure the router efficiency in terms of the query routing time (comprising of query pre-processing time, and set-cover computation time) and routing quality in terms of the query span. The query pre-processing time includes the time for query parsing, determining the tuples accessed by the query, their replicas and their locations. We study the variation of router efficiency and quality with the variation in the min-cut of the hypergraph. All plots in this section show average quantities per query over 350K TPC-C queries.

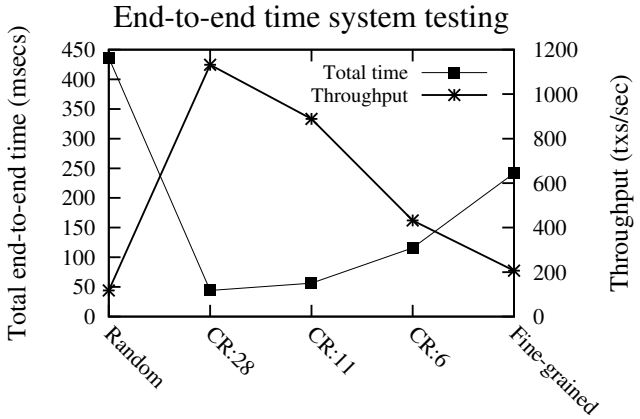
Query routing time: Figure 3.7(a) shows the comparison of query routing times for different partitioning schemes on the log scale. Random and fine-grained partitioning have much higher query pre-processing times since they require lookups into large tuple-level routing tables, whereas, the average query pre-processing time for the compressed hypergraph reduces with the increase in the compression ratio and is substantially smaller than the other two partitioning schemes. This can be attributed to lookups into much smaller hash tables making the system scalable for handling large query workloads.



(a)



(b)



(c)

Figure 3.7: **(a) Effect of hypergraph compression in minimizing the query pre-processing time and the set-cover computation time (note that the y-axis is in log scale); (b) The transaction dispatch time is directly dependent on the query span. (c) End-to-end system performance in terms of the end-to-end transaction time and the throughput for the compared schemes.**

The set-cover computation time is dependent on the size of the partition-wise list of data items accessed by the query. For tuple-level partitioning schemes, this partition-wise list is in terms of individual tuples and for the compressed graph partitioning schemes it is in terms of virtual nodes. Consequently, random and fine-grained partitioning have a much higher set-cover computation time. The set cover time decreases with increase in CR and compressed graph partitioning with the highest compression ratio (CR:28) having the lowest set-cover computation time.

Query span analysis: Figure 3.7(b) gives a comparison of the average query span for different partitioning schemes and compares its effect on the transaction dispatch time which is the time taken by the transaction manager for executing transactions on the distributed database partitions[5]. The query span is a measure of the quality of data placement achieved. Random partitioning has the highest query span and the fine-grained partitioning has the lowest, while the query spans of the compressed hypergraphs for different CRs fall between those two. The transaction dispatch time closely follows the distribution of the average query spans, wherein a higher query span results in a higher transaction dispatch time. It is pertinent to note here that the reduction in transaction dispatch time achieved using fine-grained partitioning as compared to compressed hypergraph is not as significant as the orders of magnitude reduction in routing time achieved through graph compression and an efficient routing mechanism.

End-to-end system testing: Figure 3.7(c) shows the comparison of the end-to-end transaction times and throughput measurements on 10 partitions for different partitioning

[5]It does not include the query routing time.

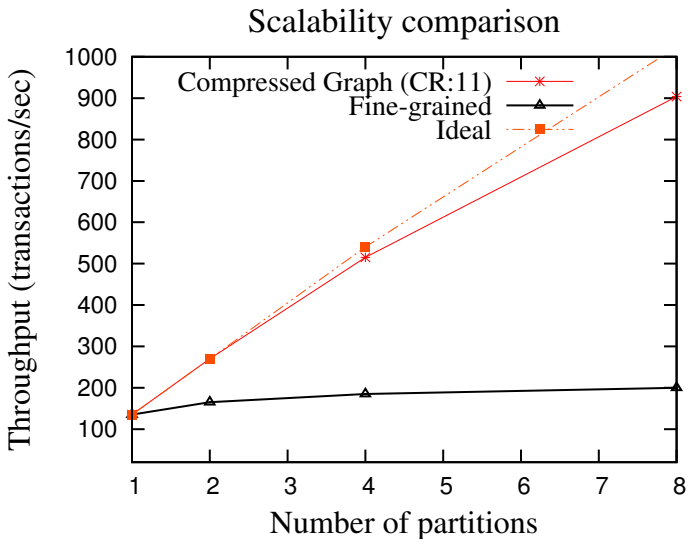


Figure 3.8: **Effect of increasing parallelism on throughput: due to the high query routing costs, the fine-grained approach is not able to effectively utilize the available parallelism.**

schemes on the log-scale. We see a substantial reduction in the total end-to-end transaction time and a high throughput for the compressed graph partitioning scheme with different CRs as compared to random and fine-grained partitioning. This can be attributed to the substantial reduction in routing time due to hypergraph compression and a reasonably good data placement as compared to fine-grained. Fine-grained does better than random due to better data placement and consequently a reduced transaction dispatch time and improved transaction throughput.

Throughput scalability: In order to test the end-to-end scalability of our system, we partitioned the workload using different partitioning schemes onto 1, 2, 4, and 8 partitions and used the TPC-C workload to ascertain the throughput as compared to an ideal linear speed-up for an embarrassingly parallel system. The results (Figure 3.8) indicate that the compressed graph scheme starts with a throughput of 165 transactions per second and achieves a throughput of 904 transactions per second for 8 partitions which is substantially

higher than that achieved by the fine-grained partitioning scheme which can be attributed to its large query processing time. The non-linear speed-up is due to contention inherent in the TPC-C workload.

3.4.6 Fine-grained Quorum Evaluation

To ascertain the suitability of different types of quorums for different transactional workloads, we used different proportions of reads and writes to generate different workload mixes. For this set of experiments, we used Mix-1, Mix-2, and Mix-3 data sets, a CR of 11 for the compressed graph, and compared its performance with random and fine-grained for different types of quorums.

We studied the variation of query span and total transaction time for ROWA (read-one-write-all) and Majority quorum for different query workload mixes. Our results (Figure 3.9) validate that for read-heavy transactional workloads, ROWA gives the minimum query spans and end-to-end transaction times, while the Majority quorum performs better for write heavy loads as they reduce the cost of distributed updates. Thus the experiments demonstrate that the choice of quorum depending on the query workload significantly impacts performance.

Figures 3.9(d) and 3.9(e) show the effect of fine-grained quorums on average query span and system throughput respectively. We experimented with different values of γ, the threshold value of $\mathcal{R}$ used by the router for deciding the type of quorum for a given node. We show the plots for $\gamma = 0.5$ which provided the best results for the TPC-C workload. Use of fine-grained quorums reduces the average query span and increases throughput

for all the partitioning strategies considered as compared to a fixed choice of ROWA or Majority for all data items, making the system adaptable to different workloads.

3.4.7 Dealing with Failures

We evaluate the effect of quorums and data placement on the ability of the system to deal with failures for the TPC-C workload. Figure 3.9(f) shows the percentage of query failures as a function of the number of partition failures. We randomly fail a given number of partitions for a given run and see its effect on the query failures. Each point on the plot is an average of 10 runs.

The results show the effectiveness of our proposed technique in terms of fault tolerance and indicate that fine-grained partitioning schemes may not perform well in presence of faults, due to a very high degree of data co-location. Our compression scheme does a relatively modest co-location of data, thereby naturally maintaining a balance between minimizing distributed transactions and providing better fault tolerance. Our experimental results show that fault tolerance improves as the CR increases. Random with ROWA provides us with a baseline to see the quantum of improvement for different data placement strategies using read one write all *available* (ROWA-A), which excludes the copies on failed partitions.

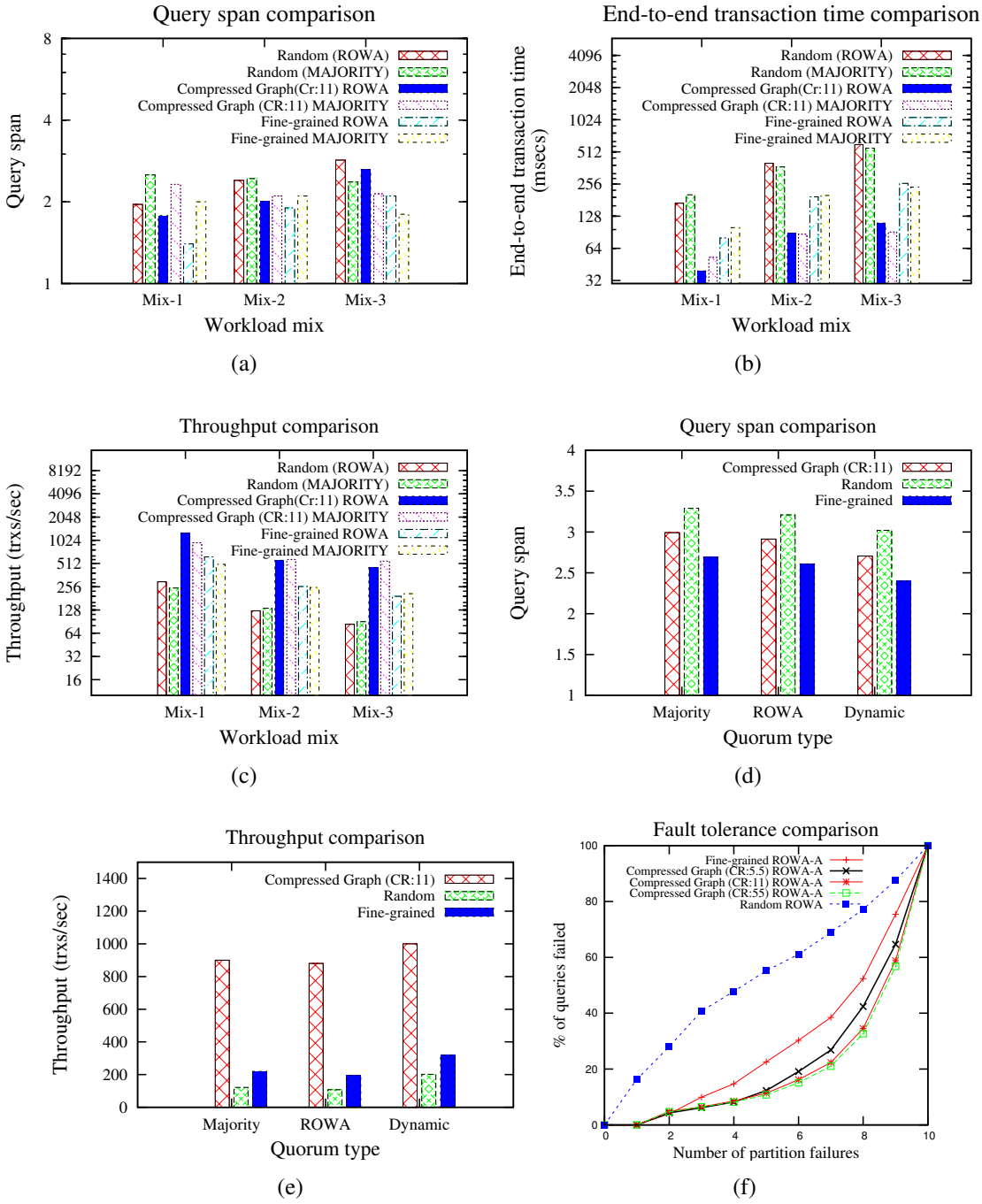


Figure 3.9: **(a)-(c) The impact of the choice of quorum on the performance for different transactional workload mixes. The different query workload mixes shown are: Mix-1 {75% read, 25%write}, Mix-2 {50% read, 50% write}, Mix-3 {25% read, 75%write}. (d)-(e) The impact of fine-grained quorums on query span and system throughput. (f) The effect of data placement on fault tolerance.**

3.5 Conclusion

In this chapter, we presented SWORD, a scalable framework for data placement and replication for supporting OLTP workloads in a cloud-based environment, that exploits available workload information. We presented a suite of techniques to reduce the cost and maintenance overheads of graph partitioning-based data placement techniques and to minimize the number of distributed transactions, while catering for fault tolerance, increased availability, and load balancing using active replication. We proposed an effective incremental repartitioning technique to maintain a good partitioning in presence of workload changes. We also explored the use of fine-grained quorums to reduce the query spans and thus improve throughputs. The use of fine-grained quorums provides our framework with the ability to seamlessly handle different workloads, an essential requirement for cloud-based environments. We have carried out an exhaustive experimental study to investigate the trade-offs between routing efficiency, partitioning time, and the quality of partitioning. Our framework provides a flexible mechanism for determining a sweet spot in terms of the compression ratio of the hypergraph, that gives a reasonably good quality of partition, improves routing efficiency, and reduces partitioning costs substantially. We have shown experimentally that our incremental repartitioning technique mitigates the impact of workload change with minimal data movement and that our proposed data placement scheme naturally improves resilience to failures.

Chapter 4: Progressive Analytics on Big Data in the Cloud

4.1 Introduction

Analytics over the increasing quantity of data stored in the Cloud has become very expensive, particularly due to the pay-as-you-go Cloud computation model. Data scientists typically manually extract samples of increasing data size (*progressive samples*) using domain-specific sampling strategies for exploratory querying. This provides them with user-control, repeatable semantics, and result provenance. However, such solutions result in tedious workflows that preclude the reuse of work across samples. On the other hand, existing *approximate query processing* systems report early results, but do not offer the above benefits for complex ad-hoc queries. A classic example is the infeasibility of sampling for join trees [64]. In these cases, a lack of user involvement with "fast and loose" progress has shortcomings; hence, data scientists tend to prefer the more laborious but controlled approach mentioned above. We illustrate this using a running example.

Example 1 (CTR)**.** Consider an advertising platform where an analyst wishes to compute the *click-through-rate* (*CTR*) for each ad. We require two sub-queries (Q_c and Q_i) to compute (per ad) the number of clicks and impressions, respectively. Each query may be non-trivial; for example, Q_c needs to process clicks on a per-user basis to consider only legitimate (non-automated) clicks from a webpage whitelist. Further, Q_i may need to

User	Ad	...
u_0	a_0	...
u_1	a_0	...
u_2	a_0	...

(a)

User	Ad	...
u_0	a_0	...
u_0	a_0	...
u_1	a_0	...
u_2	a_0	...
u_2	a_0	...

(b)

Ad	Clicks
a_0	3

Ad	Imprs
a_0	5

(c)

Ad	CTR
a_0	0.6

(d)

Figure 4.1: **(a) Click data; (b) Impression data; (c) Final result of Q_c and Q_i; (d) Final result of Q_{ctr}.**

Ad	Clicks
a_0	2
a_0	3

(a)

Ad	Imprs
a_0	1
a_0	4
a_0	5

(b)

Ad	CTR
a_0	2.0
a_0	0.5
a_0	0.6

(c)

Ad	CTR
a_0	3.0
a_0	0.75
a_0	0.6

(d)

Figure 4.2: **(a) Progressive Q_c output; (b) Progressive Q_i output; (c) & (d) Two possible progressive Q_{ctr} results.**

process a different set of logged data. The final query Q_{ctr} joins (for each ad) the results of Q_c and Q_i, and computes their ratio as the CTR. Figure 4.1 shows a toy input sorted by user, and the final results for Q_c, Q_i, and Q_{ctr}.

Next, Figure 4.2 (a) and (b) show *progressive* results for the same queries Q_c and Q_i. Without user involvement in defining progressive samples, the exact sequence of progressive counts can be non-deterministic across runs, although the final counts are precise. Further, depending on the relative speed and sequence of results for Q_c and Q_i, Q_{ctr} may compose arbitrary progressive results, resulting in significant variations in progressive CTR results. Figures 4.2(c) and (d) show two possible results for Q_{ctr}. For example, a CTR of 2.0 would result from combining the first tuple from Q_c and Q_i. Some results that are not even meaningful (e.g., CTR > 1.0) are possible. Although both results eventually get to the same final CTR, there is no mechanism to ensure that the inputs being correlated to compute progressive CTRs are deterministic and comparable (e.g., computed using the same sample of users).

The above example illustrates several challenges that are mentioned below:

1) User-Control: Data scientists usually have domain expertise that they leverage to select from a range of sampling strategies [7–9] based on their specific needs and context. In Example 1, we may progressively sample both datasets identically in user-order for meaningful progress, avoiding the join sampling problem [64]. Users may also need more flexibility; for instance, with a star-schema dataset, they may wish to *fully process* the small dimension table before sampling the fact table, for better progressive results.

2) Semantics: Relational algebra provides precise semantics for SQL queries. Given a set of input tables, the correct output is defined by the input and query alone, and is independent of dynamic properties such as the order of processing tuples. However, for complex queries, existing AQP systems use *operational semantics*, where early results are on a best-effort basis. Thus, it is unclear what a particular early result means to the user.

3) Repeatability & Optimization: Two runs of a query in AQP may provide a different sequence of early results, although they have to both converge to the same final answer. Thus, without limiting the class of queries which are progressively executed, it is hard to understand what early answers mean, or even recognize anomalous early answers. Even worse, changing the physical operators in the plan (e.g., changing operators within the ripple join family [65]) can significantly change what early results are seen!

4) Provenance: Users cannot easily establish the provenance of early results, e.g., link an early result (CTR=3.0) to particular contributing tuples, which is useful to debug and reason about results.

5) Query Composition: The problem of using operational semantics is exacerbated when one starts to compose queries. Example 1 shows that one may get widely varying results

(e.g., spurious CTR values) that are hard to reason about.

6) Scale-Out: Performing progressive analytics at scale exacerbates the above challenges. The CTR query from Example 1 is expressed as two *map-reduce* (*MR*) jobs that partition data by UserId, feeding a third job that partitions data by a different key (AdId); see Figure 4.3. In a complex distributed multi-stage workflow, accurate deterministic progressive results can be very useful. *Map-reduce-online* (*MRO*) [17] adds a limited form of pipelining to MR, but MRO reports a heuristic progress metric (average fraction of data processed across mappers) that does not eliminate the problems discussed above (Chapter 2 covers related work).

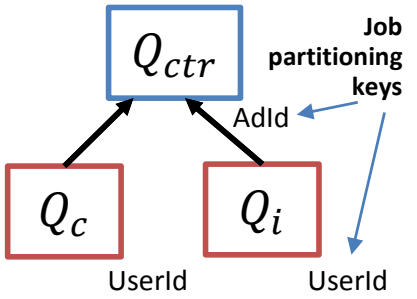


Figure 4.3: **CTR; MR jobs.**

In this dissertation we design and build a new progressive analytics system called NOW! [66]. NOW! enables fault tolerant execution of complex analysis tasks on massive amounts of data in a scaled out distributed setting and provides meaningful progressive answers to users at intervals of their choice. NOW! is particularly suitable for progressive analytics on big data in the Cloud, since it supports queries that are complex, and memory- and CPU-intensive.

NOW! is based on a progress model called PRISM(*Progressive sampling model*) [66] which enables users to encode their chosen progressive sampling strategy into the data by augmenting tuples with explicit *progress intervals* (*PIs*). PIs denote logical points where tuples enter and exit the computation, and explicitly assign tuples to progressive samples. NOW! treats these PIs as first class citizens in the framework and provides

closed-world determinism: the exact sequence of early results is a deterministic function of augmented inputs and the logical query alone. They are independent of physical plans, which enables side-effect-free query optimization. Provenance is explicit; result tuples have PIs that denote the exact set of contributing inputs. NOW! also allows meaningful query composition, as operators respect PIs. If desired, users can encode confidence interval computations as part of their queries. PIs offer remarkable flexibility for encoding sampling strategies and ordering for early results, including arbitrarily overlapping sample sequences and special cases such as the star-schema join (Chapter 4 provides more details).

NOW! generalizes the popular data-parallel MR model and supports *progress-aware reducers* that understand explicit progress in the data. Instead of modifying an existing relational engine to support progressive analytics, we use an unmodified temporal streaming engine, by carefully reinterpreting its temporal fields to denote progress. In particular, NOW! uses StreamInsight [67] as a progress-aware reducer to enable scaled-out progressive relational (SQL) query support in the Cloud. Provision of meaningful early results on large volumes of data using significantly fewer resources, substantially reduces the cost of data analytics in the Cloud. We also extend NOW! with a high performance mode that eliminates disk writes, and discuss high availability (by leveraging progress semantics in a new way) and straggler management. We perform a detailed evaluation of NOW! in a cloud setting over real and benchmark datasets up to 100GB, with up to 75 large-sized Windows Azure compute instances. Experiments show that we can scale effectively and produce meaningful early results, making NOW! suitable in a pay-as-you-go environment. NOW! provides a substantial reduction in processing time, memory and CPU usage

PI	User	Ad
$[0, \infty)$	u_0	a_0
$[1, \infty)$	u_1	a_0
$[2, \infty)$	u_2	a_0

(a)

PI	User	Ad
$[0, \infty)$	u_0	a_0
$[0, \infty)$	u_0	a_0
$[1, \infty)$	u_1	a_0
$[2, \infty)$	u_2	a_0
$[2, \infty)$	u_2	a_0

(b)

PI	Ad	Clicks
$[0, 1)$	a_0	1
$[1, 2)$	a_0	2
$[2, \infty)$	a_0	3

PI	Ad	Imprs
$[0, 1)$	a_0	2
$[1, 2)$	a_0	3
$[2, \infty)$	a_0	5

(c)

PI	Ad	CTR
$[0, 1)$	a_0	0.5
$[1, 2)$	a_0	0.66
$[2, \infty)$	a_0	0.6

(d)

Figure 4.4: **(a,b) Input data with progress intervals; (c) Progressive results of Q_c and Q_i; (d) Progressive output of Q_{ctr}.**

as compared to current schemes; performance is significantly enhanced by exploiting sort orders and using our memory-only processing mode.

Outline. We begin with a background on PRISM in Section 4.2. We then discuss NOW! in detail in Section 4.3; and discuss several extensions in Section 4.4. The detailed evaluation of NOW! is covered in Section 4.5 and finally Section 4.6 concludes this section of the proposal.

4.2 Background

In this section, I provide a detailed background on PRISM, the progress model used by NOW!.

4.2.1 PRISM semantics & construction

At a high level, PRISM defines a logical linear *progress domain* that represents the progress of a query. Sampling strategies desired by data scientists are encoded into the data before query processing, using *augmented tuples with progress intervals* that precisely define how data progressively contributes to result computation. Users express their data analytics as relational queries that consist of a DAG of *progressive operators*. An

extension of traditional database operators, progressive operators understand and propagate progress intervals based on precisely defined operator semantics. The result of query processing is a sequence of augmented tuples whose progress intervals denote *early results and their associated regions of validity in the progress domain*. Each of these steps is elaborated in the following subsections.

4.2.2 Logical Progress and Progress Intervals

PRISM defines a logical linear *progress domain* $\mathcal{P}$ as the range of non-negative integers $[0, \infty)$. Progress made by a query at any given point during computation is explicitly indicated by a non-decreasing *progress point* $p \in \mathcal{P}$. Progress point ∞ indicates the point of query completion. Next, we associate a *progress interval* (*PI*) from the progress domain to every tuple in the input data. More formally, each tuple T is augmented with two new attributes, a *progress-start* P^+ and a *progress-end* P^-, that jointly denote a PI $[\mathsf{P}^+, \mathsf{P}^-)$. P^+ indicates the progress point at which a tuple T starts participating in the computation, and P^- (if not ∞) denotes the progress point at which tuple T stops contributing to the computation. PIs enable users to specify domain-specific progressive sampling strategies. PI assignment can be controlled by data scientists to ensure quicker and more meaningful early results, either directly or using a layer between the system and the user. Figures 4.4(a) and (b) show PIs for our running example inputs; they are also depicted in Figure 4.5 (top). We provide several concrete examples of PI assignment in Section 4.2.6.

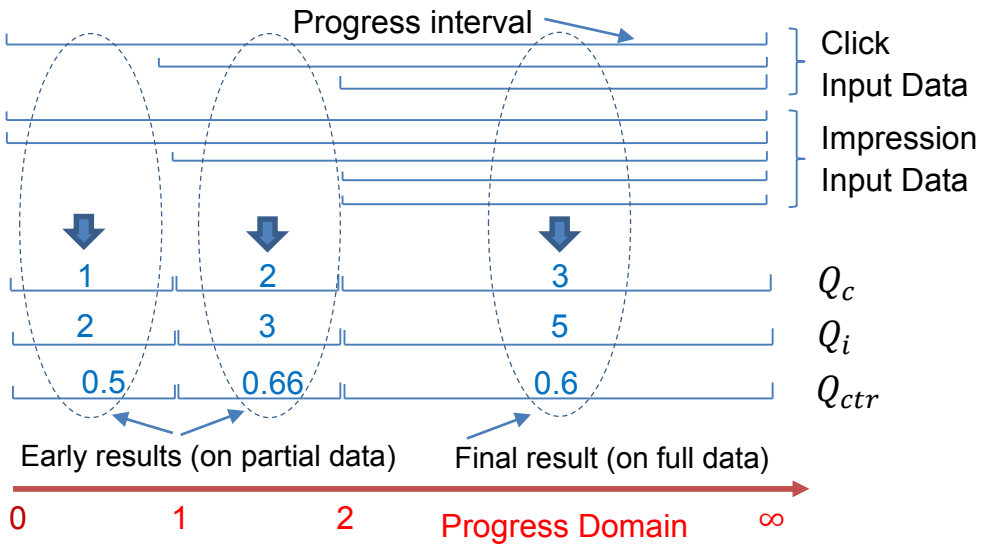


Figure 4.5: **Input and output progress intervals, query semantics.**

4.2.3 Progressive Operators and Queries

Progressive Operators Every relational operator $\mathcal{O}$ has a progressive counterpart, which computes augmented output tuples from augmented input tuples. Logically, the output at progress point p is the operation $\mathcal{O}$ applied to input tuples whose PIs are stabbed by p. Figures 4.4(c) and 4.5 show the results of Q_c and Q_i, which behave as Count operators. We see that Q_c produces a progressive count of 1 at progress point 0, which it revises to 2 and 3 at progress points 1 and 2. As a result, the PIs for these tuples are $[0, 1)$, $[1, 2)$ and $[2, \infty)$ respectively. The P^- for an output tuple may not always be known at the same time as when the operator determine its P^+. Thus, an operator may output a tuple having an eventual PI of $[\mathsf{P}^+,\mathsf{P}^-)$ in two separate pieces: (1) at progress point P^+, it generates a *start-edge* tuple T_1 with a PI $[\mathsf{P}^+, \infty)$ indicating that the tuple participates in the result forever; (2) at the later progress point P^-, it generates an *end-edge* tuple T_2 with the actual PI $[\mathsf{P}^+,\mathsf{P}^-)$. We use the term *progress-sync* to denote the progress point associated with

a tuple (or its subsequent update). The start-edge tuple T_1 has a progress-sync of P^{+}, whereas the end-edge tuple T_2 has a progress-sync of P^{-}.

Every operator both processes and generates augmented tuples in *non-decreasing progress-sync order*. The eventual P^{-} values for early results that get refined later are less than ∞, to indicate that the result is not final. For example, consider an Average operator that reports a value a_0 from progress point 0 to 10, and revises it to a_1 from progress point 10 onwards. Tuple a_0 has an eventual PI of $[0, 10)$. This is reported as a start-edge $[0, \infty)$ at progress point 0. At progress point 10, the operator reports an end-edge $[0, 10)$ for the old average a_0, followed immediately by a start-edge $[10, \infty)$ for the revised average a_1. Similarly, a progressive Join operator with one tuple on each input with PIs $[10, 20)$ and $[15, 25)$ – if the join condition is satisfied – produces a result tuple with PI $[15, 20)$, the intersection of the two input PIs. Note here that the output tuple's PI ends at 20 because its left input is no longer valid at that point.

Progressive Queries Based on the above semantics, operators can be composed meaningfully to produce progressive queries. We define PRISM output for a relational query Q as:

Definition 1 (PRISM Output)**.** Associated with each input tuple is a progress interval (PI). At every unique progress point p across all PI endpoints in the input data, there exists a set $\mathcal{O}_p$ of output results with PIs stabbed by p. $\mathcal{O}_p$ is defined to be exactly the result of the query Q evaluated over input tuples with PIs stabbed by p.

4.2.4 Summary of Benefits of the PRISM Model

The results of Q_{ctr} for our running example are shown in Figures 4.4(d) and 4.5; every CTR is meaningful as it is computed on some prefix of users (for our chosen progress assignment), and CTR provenance is provided by PIs. The final CTR of 0.6 is the only tuple active at progress point ∞, as expected.

It is easy to see that the output of a progressive query is a deterministic function of the (augmented) input data and the logical query alone. Further, these progressive results are fixed for a given input and logical query, and are therefore repeatable. PRISM enables data scientists to use their domain knowledge to control progressive samples; Section 4.2.6 provides several concrete examples. Early results in PRISM carry the added benefit of provenance that helps debug and reason about early results: the set of output tuples with PIs stabbed by progress point p denote the progressive result of the query at p. The provenance of these output tuples is simply all tuples along their input paths whose PIs are stabbed by p.

One can view PRISM as a generalization of relational algebra with progressive sampling as a first-class concept. Relational algebra prescribes the final answer to a relational query but does not cover how we get there using partial results. The PRISM algebra explicitly specifies, for any query, not only the final answer, but every intermediate (progressive) result and its position in the progress domain.

4.2.5 Implementing PRISM

One can modify a database engine to add PI support to all operators in the engine. However, we can realize PRISM *without* incurring this effort. The idea is to leverage a *stream processing engine* (*SPE*) as the progressive query processor. In particular, the semantics underlying a temporal SPE such as NILE [68], STREAM [69], or StreamInsight [67] (based on temporal databases [70]) can be leveraged to denote progress, with the added benefit of incremental processing across samples when possible. With StreamInsight's temporal model, for example, the event validity time interval [71] $[V_s, V_e)$ directly denotes the PI [P^+,P^-). T_1 is an insertion and T_2 is a retraction (or revision [72]). Likewise, T_1 and T_2 correspond to Istreams and Dstreams in STREAM, and positive and negative tuples in NILE. We feed the input tuples converted into events to a *continuous query* corresponding to the original atemporal SQL query. The unmodified SPE operates on these tuples as though they were temporal events, and produces output events with timestamp fields that we re-interpret as tuples with PIs.

Note that with this construction, the SPE is unaware that it is being used as a progressive SQL processor. It processes and produces events whose temporal fields are re-interpreted to denote progress of an *atemporal* (relational) query. For instance, the temporal symmetric-hash-join in an SPE effectively computes a sequence of joins over a sequence of progressive samples very efficiently. The resulting query processor transparently handles all of SQL, including user-defined functions, with all the desirable features of our new progress model.

4.2.6 PI Assignment

Any progressive sampling strategy at the inputs corresponds to a PI assignment; several are discussed next.

Inclusive & Non-inclusive Samples With *inclusive samples* (as used, for example, in EARL [36]), each sample is a superset of the previous one. To specify these, input tuples are assigned a P^{-} of ∞, and non-decreasing P^{+} values based on when tuples become a part of the sample, as shown in Figure 4.5 (top). In case of *non-inclusive samples*, tuples have a finite P^{-} to denote that they no longer participate in computation beyond P^{-}, and can even reappear with a greater P^{+} for a later sample (our technical report [38] includes a concrete example of expressing non-inclusive sampling using PIs).

Reporting Granularity Progress reporting granularity can be controlled by individual queries, by adjusting the way P^{+} moves forward. Data is often materialized in a statistically relevant order, and we may wish to include k additional tuples in each successive sample. We use a streaming AlterLifetime [73] operator that sets P^{+} for the n^{th} tuple to $\lfloor n/k \rfloor$ and P^{-} to ∞. This increases P^{+} by 1 after every k tuples, resulting in the engine producing a new progressive result every k tuples. We refer to the set of tuples with the same P^{+} as a *progress-batch*. Data scientists often start with small progress-batches to get quick estimates, and then increase batch sizes (e.g., exponentially) as they get diminishing returns with more data.

Joins & Star Schemas In case of queries involving an equi-join, we may apply an identical sampling strategy (e.g., pseudo-random) over the join key in both inputs as this

increases the likelihood of getting useful early results. With a star-schema, we may set all tuples in the small dimension table to have a PI of $[0, \infty)$, while progressively sampling from the fact table as $[0, \infty), [1, \infty), \ldots$. This causes a Join operator to "preload" the dimension table before progressively sampling the fact table for meaningful early results.

Stratified Sampling Stratified sampling groups data on a certain key and applies a sampling strategy (e.g., uniform) within each group to ensure that rare subgroups are sufficiently represented. BlinkDB [37] pre-computes stratified samples of different sizes and responds to queries within a given error and response time by choosing the correct sample to compute the query on. Stratified sampling is easy to implement with PRISM: we perform a GroupApply operation [73] by the key, with an AlterLifetime inside the GroupApply to create progress-batches as before. The temporal Union that merges groups respects timestamp ordering, resulting in a final dataset with PIs that exactly represent stratified sampling. Stratified samples of increasing size can be constructed similarly.

Other Examples For online aggregation, we may assign non-decreasing P^+ values over a pre-defined random order of tuples for quick result convergence. Active learning [7] changes the sampling strategy based on outcomes from prior samples. Prior proposals for ordering data for quick convergence [13, 15, 35, 65] simply correspond to different PI assignment schemes in PRISM.

4.2.7 Performance Optimizations

Query processing using an in-memory streaming engine can be expensive since the final answer is over the entire dataset. PRISM enables crucial performance optimizations

that can improve performance significantly in practical situations. Consider computation Q_c, which is partitionable by UserId. We can exploit the compile-time property that progress-sync ordering is the same as (or correlated to) the partitioning key, to reduce memory usage and consequently throughput. The key intuition is that although every tuple with PI $[\mathsf{P}^+, \infty)$ logically has a P^- of ∞, it does not contribute to any progress point beyond P^+. Thus, we can temporarily set P^- to $\mathsf{P}^+ + 1$ before feeding the tuples to the SPE. This effectively causes the SPE to not have to retain information related to progress point P^+ in memory once computation for P^+ is done. The result tuples have their P^- set back to ∞ to retain the original query semantics (these query modifications are introduced using compile-time query rewrites). A similar optimization applies to equi-joins; see [38] for details. We will see in Section 4.5 that this optimization can result in orders-of-magnitude performance benefits.

4.3 NOW! Architecture and Design

4.3.1 Overview

At a high level, NOW!'s architecture is based on the Map-Reduce (MR) [74] computation paradigm. Figure 4.6 shows the overall design of NOW! (right) as compared to vanilla MR (left), for a query with two stages and different partitioning keys. *Blobs* in the figure indicate the format of input and output data on Windows Azure's distributed Cloud storage, and can be replaced by any distributed persistent storage such as HDFS. The key points are as follows:

1) Progress-aware data flow: NOW! implements the PRISM progress model and provides

support for data flow (§ 4.3.2) in strict progress-sync order. The main components of progress-aware data flow are:

- **Batching** NOW! reads input data annotated with PIs (progressive samples) and creates *batches* (§ 4.3.2.1) of tuples with the same progress-sync. Data movement in NOW! is fully pipelined in terms of these *progress-batches*, in progress-sync order.

- **Sort-free data shuffle** MR sorts the map output by key, followed by a merge to enable grouping by key at reducers. This sort-merge operation in MR is a performance bottleneck [39]. In contrast, the batched map output in NOW! is partitioned and shuffled across the network to reducers *without sorting* (§ 4.3.2.2), thus retaining progress-sync order with improved performance.

- **Progress-aware merge** A progress-aware merge at reducers is key to enabling the PRISM model for progressive query results. Each reducer groups together batches received from different mappers, that belong to the same PI, into a single progress-batch, and ensures that all progress-batches are processed in strict progress-sync order (§ 4.3.2.3) along all data flow paths.

Data flow between map and reduce in NOW! uses TCP connections which guarantee FIFO delivery. Since the input data is read in progress-sync order and all components retain this invariant, we are guaranteed global progress-sync order for progress-batches.

2) Progress-aware reducers: NOW! introduces the notion of a *progress-aware reducer* (Section 4.3.2.4), that accepts and produces augmented tuples in progress-sync order,

and logically adheres to the PRISM query model. The progress-aware merge generates progress-batches in progress-sync order; these are fed directly to reducers that produce early results in progress-sync order. While one could write custom reducers, we use an unmodified SPE (§ 4.2.5) as a progress-aware reducer for progressive relational queries.

3) Multi-stage support: NOW! supports *concurrent scheduling* of all jobs in a multi-stage query and *co-location* of mappers of dependent jobs with the reducers of feeding jobs on the same slave machine (Section 4.3.3). Data transfer between jobs is in-memory providing significant savings in a Cloud deployment where blob access is expensive.

4) Flow control: NOW! provides end-to-end flow control to avoid buffer overflows at intermediate stages such as mapper output, reducer input and reducer output for multi-stage MR. The flow control mechanism ensures data flows at a speed that can be sustained by downstream consumers. We use a *blocking concurrent queue* (*BCQ*), a lock-free data structure which supports concurrent enqueue and dequeue operations, for implementing an end-to-end flow control mechanism for NOW! (our technical report [38] has more details on flow control in NOW!).

5) In-memory data processing: By default, NOW! materializes map output on disk to provide better data availability during failure recovery. For better interactivity, we also support a high-performance in-memory mode (see Section 4.4).

4.3.2 Progress-aware Data Flow & Computation

Data flow in NOW! is at the granularity of progress-batches and governed by PIs. This section describes the generation and flow of these progress-batches in the framework.

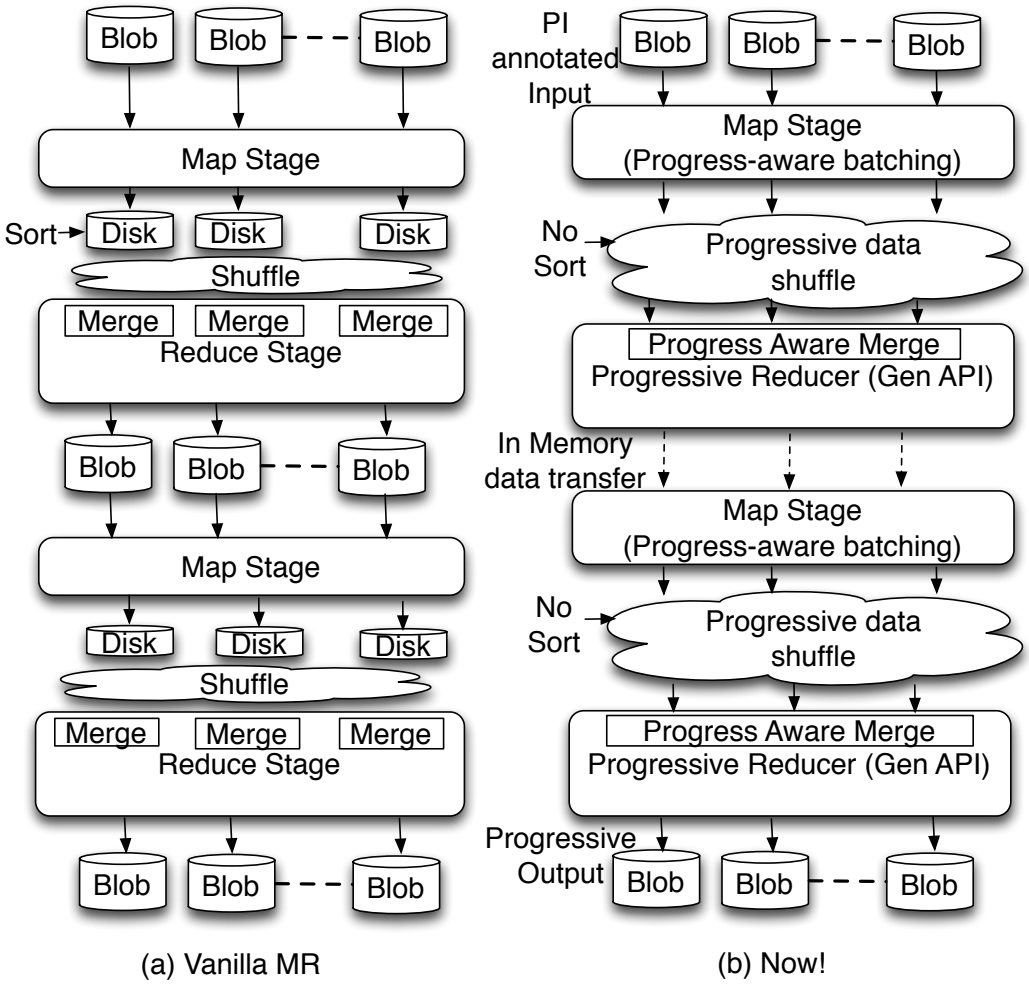


Figure 4.6: **System architecture (MR vs. NOW!).**

4.3.2.1 Progress-aware Batching

The input data is partitioned into a number of input splits (one for each mapper), data tuples in each of which are assigned progress intervals in progress-sync order. The mapper reads its input split as progress annotated tuples (progressive samples), and invokes the user's map function.The resulting augmented key-value pairs are partitioned by key to produce a sequence of *progress-batches* for each partition (downstream reducer).

A progress-batch consists of all tuples with the same progress-sync value (within the specific partition) and has a unique ID. Each progress-batch sequence is in strictly increasing progress-sync order. The input text reader appends an *end-of-file* (*eof*) marker to the mapper's input when it reaches the end of its input split. The mapper, on receipt of the *eof* marker, appends it to all progress-batch sequences.

Batching granularity. The batching granularity in the framework is determined by the PI assignment scheme (§ 4.2.6) of the input data. NOW!, also provides a *control knob* to the user, in terms of a *parameterized batching function*, to vary the batching granularity of the map output as a factor of the PI annotation granularity of the actual input. This avoids re-annotating the input data with PIs if the user decides to alter the granularity of the progressive output.

Example 2 (Batching)**.** Figure 4.7(a) shows a PI annotated input split with three progressive samples. Figure 4.7(b) shows the corresponding batched map output, where each tuple in a batch has the same progress-sync value. Figure 4.7(c) shows how progress granularity is varied using a batching function that modifies P^+. Here, $P^+ = \lfloor \frac{P^+}{b} \rfloor$ is the batching function, with the batching parameter b set to 2.

4.3.2.2 Progressive Data Shuffle

NOW! shuffles data between the mappers and reducers in terms of progress-batches without sorting. As an additional performance enhancement, NOW! supports a mode for in-memory transfer of data between the mappers and reducers with flow control to avoid memory overflow. We pipeline progress-batches from the mapper to the reducers using

PI	User	Ad
$[0, \infty)$	u_0	a_0
$[0, \infty)$	u_0	a_0
$[1, \infty)$	u_1	a_0
$[1, \infty)$	u_1	a_1
$[2, \infty)$	u_2	a_1
$[2, \infty)$	u_2	a_1
$[2, \infty)$	u_2	a_1
$[2, \infty)$	u_2	a_1

(a)

PI	User	Ad
$[0, \infty)$	u_0	a_0
$[0, \infty)$	u_0	a_0
PI	User	Ad
$[1, \infty)$	u_1	a_0
$[1, \infty)$	u_1	a_1
PI	User	Ad
$[2, \infty)$	u_2	a_1
$[2, \infty)$	u_2	a_1
$[2, \infty)$	u_2	a_1
$[2, \infty)$	u_2	a_1

(b)

PI	User	Ad
$[0, \infty)$	u_0	a_0
$[0, \infty)$	u_0	a_0
$[0, \infty)$	u_1	a_0
$[0, \infty)$	u_1	a_1
PI	User	Ad
$[1, \infty)$	u_2	a_1
$[1, \infty)$	u_2	a_1
$[1, \infty)$	u_2	a_1
$[1, \infty)$	u_2	a_1

(c)

Figure 4.7: **(a) Input data annotated with PIs; (b) Progress-batches according to input data PI assignment; (c) Progress-batches with modified granularity using a batching function.**

a *fine-grained signaling mechanism*, which allows the mappers to inform the job tracker (master) the availability of a progress-batch. The job tracker then passes the progress-batch ID and location information to the appropriate reducers, triggering the respective map output downloads.

The download mechanism on the reducer side has been designed to support progress-sync ordered batch movement. Each reducer maintains a separate blocking concurrent queue (BCQ) for each mapper associated with the job. As mentioned earlier, the BCQ is a lock-free in-memory data structure which supports concurrent enqueue and dequeue operations and enables appropriate flow control to avoid swamping of the reducer. The maximum size of the BCQ is a tunable parameter which can be set according to the available memory at the reducer . The reducer enqueues progress-batches, downloaded from each mapper, into the corresponding BCQ associated with the mapper, in strict progress-sync order. Note that our batched sequential mode of data transfer means that continuous connections do not need to be maintained between mappers and reducers, which aids scalability.

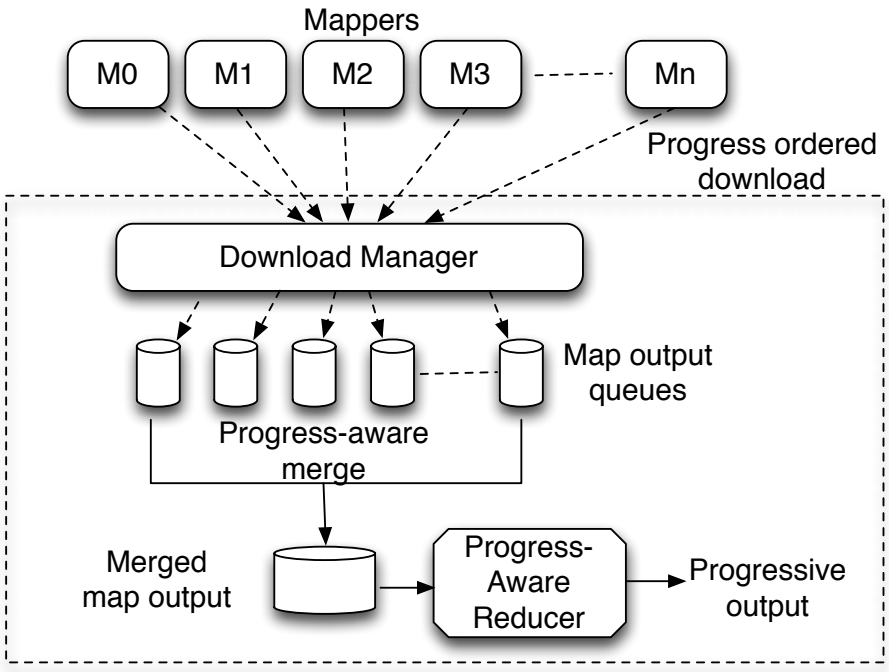


Figure 4.8: **Progress-aware merge.**

4.3.2.3 Progress-aware Merge

NOW! implements the PRISM model using a *progress-aware merge mechanism* which ensures flow of data in progress-sync order along all paths in the framework. Figure 4.8 shows the high level design of the progress-aware merge module within each reducer. Once a map output is available in each of the map output queues, the reducer invokes the progress-aware merge mechanism the details of which are given in Algorithm 3.

The algorithm takes as input the number of mappers M, a set of BCQs $\mathcal{B}$ where $q_i \in \mathcal{B}$ denotes the blocking concurrent queue for mapper i, the current progress-sync value c_{min} of the merged batch that needs to be produced (c_{min} is initialized to the minimum progress-sync across the heads of the BCQs), and $\mathcal{H}$, where $h_i \in \mathcal{H}$ is the progress-sync value currently at the head of q_i (h_i is initialized to the progress-sync value at the head of

Algorithm 3: Progress-aware merge

```
input : # of Mappers M, B = {q_1, ..., q_M}, c_min, H = {h_1, ..., h_M}
output: Merged batch O
begin
    O = ∅;
    for each q_i ∈ Q do
        if (h_i==∞) then continue;
        ;
        progress-sync = peek(q_i);  // peek blocks if q_i = ∅
        if (progress-sync==eof) then
            h_i = ∞; continue;
        end
        h_i =progress-sync;
        if (h_i == c_min) then
            O = O ⋃ dequeue(q_i);
            progress-sync = peek(q_i);
            if (progress-sync==eof) then h_i = ∞;
            ;
            else h_i =progress-sync;
            ;
        end
    end
    c_min = min(H); return O;
end
```

q_i).

The algorithm initializes an empty set $\mathcal{O}$ as output. It iterates over all mapper queues to find and dequeue the batches whose progress-sync values match c_{min}, adds them to $\mathcal{O}$ and updates h_i to the new value at the head of q_i. It finally updates c_{min} and returns $\mathcal{O}$, a merged batch with all tuples having the same progress-sync value. $\mathcal{O}$ is then fed to the progressive reducer. If $\mathcal{O} = \emptyset$, indicating end of input on all BCQs, the framework passes an *eof* marker to the progressive reducer signaling termination of input.

4.3.2.4 Progress-aware Reducer

Let *partition* denote the set of keys that a particular reducer is responsible for. In traditional MR, the reducer gathers all values for each key in the partition and invokes a reduce function for each key, passing the group of values associated with that key. NOW! instead uses progress-aware reducers whose input is a sequence of progress-batches associated with that partition in progress-sync order. The reducer is responsible for per-key grouping and computation, and produces a sequence of progress-batches in progress-sync order as output. We use the following API to achieve this:

```
Unchanged map API:
void map(K1 key, V1 value, Context context

Generalized Reduce API:
void reduce( Iterable<K2, V2> input, Context context)
```

Here, `V1` and `V2` include PIs. NOW! also supports the traditional reducer API to support older workflows, using a layer that groups active tuples by key for each progress point, invokes the traditional reduce function for each key, and uses the reduce output to generate tuples with PIs corresponding to that progress point.

Progressive SQL While users can write custom progress-aware reducers, we advocate using an unmodified temporal streaming engine (such as StreamInsight) as a reducer to handle progressive relational queries (§ 4.2.5). Streaming engines process data in timestamp order, which matches with our progress-sync ordered data movement. Temporal notions in events can be reinterpreted as progress points in the query. Further, streaming engines naturally handle efficient grouped subplans using hash-based key partitioning,

which is necessary to process tuples in progress-sync order.

4.3.3 Support for Multi-stage

We find that most analytics queries need to be expressed as multi-stage MR jobs. NOW! supports a fully pipelined progressive job execution across different stages using concurrent job scheduling and co-location of processes that need to exchange data across jobs.

Concurrent Job Scheduling The scheduler in NOW! has been designed to receive all the jobs in a multi-stage query as a job graph, from the application controller. Each job is converted into a set of map and reduce tasks. The scheduler extracts the type information from the job to construct a dependency table that tracks, for each task within each job, where it reads from and writes to (a blobs or some other job). The scheduler uses this dependency table to partition map tasks into a set of *independent* map tasks M_i which read their input from a blob/HDFS, and a set of *dependent* map tasks M_d whose input is the output of some previous stage reducer. Similarly, reduce tasks are partitioned into a set of *feeder* tasks R_f that provide output to mappers of subsequent jobs, and a set of *output* reduce tasks R_o that write their output to a blob/HDFS.

Algorithm 4 shows the details of how the map and reduce tasks corresponding to different jobs are scheduled[1]. First, all the reduce tasks in R_f are scheduled on slave machines that have at least one map slot available to schedule a corresponding dependent map task in M_d which would consume the feeder reduce task's output. The scheduler

[1] If the scheduler is given additional information such as the streaming query plan executing inside reducers, we may be able to leverage database cost estimation techniques to improve the scheduling algorithm. This is a well studied topic in prior database research, and the ideas translate well to our setting.

Algorithm 4: Scheduling

```
input : R_f, R_o, M_i, M_d, dependency table
begin
    for each r ∈ R_f do
        Dispatch r;
        if Dispatch successful then Make a note of tracker ID;
        ;
    end
    for each r ∈ R_o do Dispatch r;
    ;
    for each m ∈ M_d do
        Dispatch m, co-locating it with its feeder reducer;
    end
    for each m ∈ M_i do
        Dispatch m closest to input data location;
    end
end
```

maintains a state of the task tracker IDs of the slave machines on which these feeder reduce tasks have been scheduled. Second, all the reducers in R_o are scheduled depending on the availability of reduce slots on various slave machines in a round robin manner. Third, all the map tasks in M_d are dispatched, co-locating them with the reducers of the previous stage in accordance with the dependency table and using the task tracker information retained in step 1 of the algorithm. Finally, all the map tasks in M_i are scheduled closest to the input data location. Placing tasks in this order ensures that if there exists a feasible placement of all MR tasks that would satisfy all job dependencies, we will find such a placement.

Data flow between jobs Figure 4.9 shows a sample placement of map and reduce tasks for processing a query that constitutes three jobs, J_1, J_2 and J_3.

Figure 4.9(a) shows the data flow between jobs and Figure 4.9(b) shows the place-

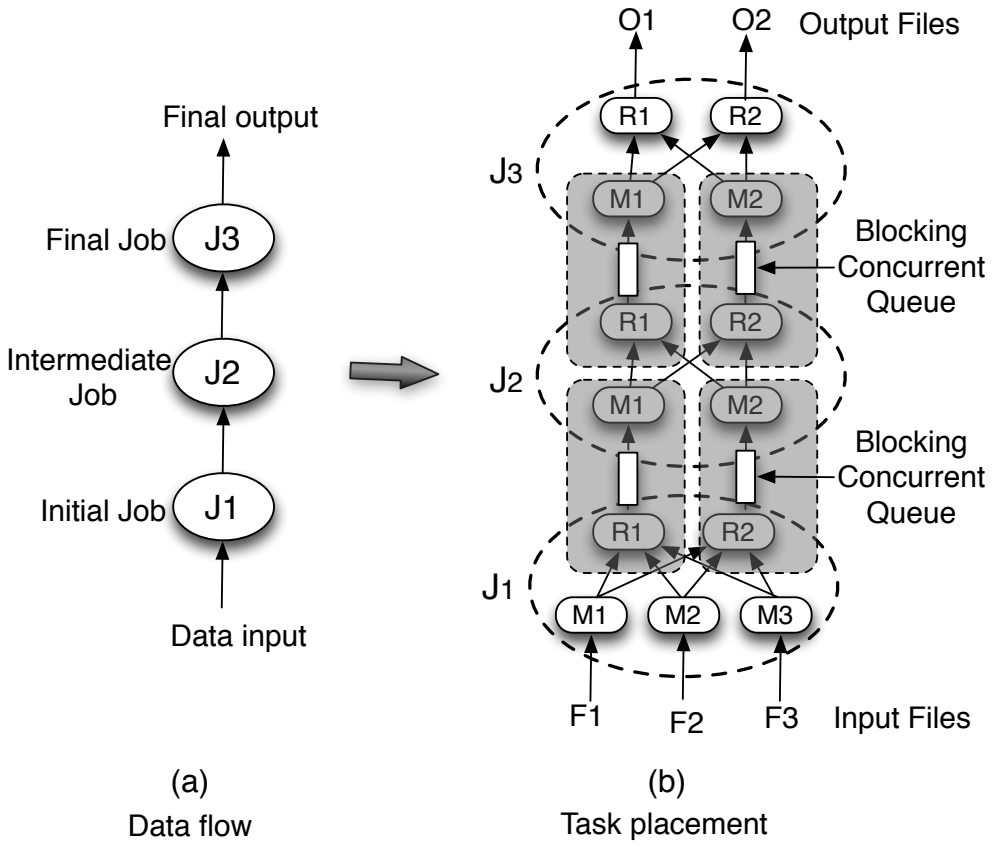


Figure 4.9: **Multi-stage map reduce data flow.**

ment of map and reduce tasks as per NOW!'s scheduling algorithm (Ref Algorithm 4). The shaded portions in the figure indicate that the corresponding map and reduce tasks have been co-scheduled on the same slave machine. The scheduler also verifies that the number of dependent map tasks are equal to the number of feeder reduce tasks of a preceding job, thus ensuring that there is one dependent map task for each feeder reduce task that is co-scheduled on the same slave machine.

Data flow between jobs is modeled on the producer-consumer paradigm using a BCQ and takes place completely in memory avoiding data materialization and shuffling overheads. Further, co-location of the reducers and mappers of dependent jobs does away with the overhead of data serialization, de-serialization and expensive network I/O be-

tween stages in a Cloud setting.

4.4 Discussion and Extensions

4.4.1 High availability (HA)

Upadhyaya et al. [75] have recently shown how a multi-stage pipelined map-reduce system can support hybrid strategies of replay and checkpointing; these solutions are applicable in our setting. Specifically, the failure semantics for NOW! are:

Map task failure: Any map task in progress or completed on a failed worker node needs to be rescheduled as in vanilla MR.

Reduce task failure: After a reduce task fails, one can replay its input starting from the last checkpoint (map output is materialized on local storage to allow replay). Interestingly, PRISM can further reduce the cost of replay after a failure. The key insight is that processing at progress point p depends only on input tuples whose PIs are stabbed by p. We can leverage this in two ways:

- We can filter out tuples with $\mathsf{P}^{-} \leq p$ during replay to significantly reduce the amount of data replayed and prune the intermediate map output saved on local storage[2].
- During replay, we can set $\mathsf{P}^{+} = \max(p, \mathsf{P}^{+})$ for replayed tuples so that the reducer does not need to re-generate early results for progress points earlier than p.

Prior research [76] has reported that input sizes on production clusters are usually less than 100GB. Further, progressive queries are usually expected to end early. There-

[2]This optimization does not apply to external input which has P^{-} set to ∞, but can apply to intermediate results in multi-stage jobs.

fore, NOW! supports an efficient *no-HA* mode, where intermediate map output is not materialized on local storage and no checkpointing is done. This requires a failure to cascade back to the source data (we simply restart the job). Restarting the job on failure is a cheap and practical solution for such systems as compared to traditional long-running jobs. That said, we acknowledge that high availability with low recovery time (e.g., by restarting only the failed parts of the DAG) is important in some cases. Prior work [44,75] has studied this problem; these ideas apply in our setting. We leave the implementation and evaluation of such fine-grained HA in NOW! as future work.

4.4.2 Straggler and Skew Management

Stragglers A consequence of progress-sync merge is that if a previous task makes slow progress, we need to slow down overall progress to ensure global progress-sync order. While progress-sync order is necessary to derive the benefits of PRISM, there are fixes that avoid sacrificing semantics and determinism:

- Consider n nodes with 1 straggler. If the processing skew is a result of imbalanced load, we can dynamically move partitions from the straggler to a new node (we need to also move reducer state). We may instead fail the straggler altogether and re-start its computation by partitioning its load equally across the remaining $n-1$ nodes. The catch-up work gets done $n-1$ times faster, resulting in a quicker restoration of balance [3].

- We could add support for *compensating reducers*, which can continue to process

[3] If failures occur halfway through a job on average, jobs run for $2.5/(n-1)$ times as long due to a straggler with this scheme.

new progress points, but maintain enough information to revise or compensate their state once late data is received. Several engines have discussed support for compensations [71, 72], and fit well in this setting.

As we have not found stragglers to be a problem in our experiments on Windows Azure VMs, the current version of NOW! does not address this issue. A deeper investigation is left as future work.

Data Skew Data skew can result from several reasons:

- Some sampling strategies encoded using PIs may miss out on outliers or rare subpopulations within a population. This can be resolved using stratified sampling which can be easily implemented in PRISM as discussed in Section 4.2.6.

- Skew in the data may result in some progress-batches being larger than others at the reducers. However, this is no different from skew in traditional map-reduce systems, and solutions such as [77] are applicable here.

Since skew is closely related to the straggler problem, techniques mentioned earlier for stragglers may also help mitigate skew.

4.5 Evaluation

4.5.1 Implementation Details

NOW! is written in C# and deployed over Windows Azure. NOW! uses the same master-slave architecture as Hadoop [78] with JobTracker and TaskTracker nodes. TaskTracker nodes are allocated a fixed number of map and reduce slots. Heartbeats are used to

ensure that slave machines are available. We modified and extended this baseline to incorporate our new design features (see Section 4.3) such as pipelining, progress-based batching, progress-sync merge, multi-stage job support, concurrent job scheduling, etc. NOW! deployed on the Windows Azure Cloud platform, uses Azure blobs as persistent storage and Azure VM roles as JobTracker and TaskTracker nodes. Multi-stage *job graphs* are generated by users and provided to NOW!'s JobTracker as input; each job consists of input files, a partitioning key (or mapper), and a progressive reducer. Although NOW! has been developed in C# and evaluated on Windows Azure, its design features are not tied to any specific platform. For example, NOW! could be implemented over Hadoop using HDFS and deployed on the Amazon EC2 cloud.

NOW! makes it easy to employ StreamInsight as a reducer for progressive SQL, by providing an additional API that allow users to directly submit a graph of ⟨key, query⟩ pairs, where *query* is a SQL query specified using LINQ [79]. Each node in this graph is automatically converted into a job. The job uses a special progressive reducer that uses StreamInsight to process tuples. The NOW! API can be used to build front-ends that automatically convert larger Hive, SQL, or LINQ queries into job graphs. Although the system has been designed for the Cloud and uses Cloud storage, it also supports deployment on a cluster of machines (or private Cloud). NOW! includes diagnostics for monitoring CPU, memory, and I/O usage statistics. These statistics are collected by an instance of a log manager running on each machine which outputs these in the form of logs which are stored as blobs in a separate container.

4.5.2 Experimental Setup

System Configuration The input and final output of a job graph are stored in Azure blobs. Each Azure VM role (instance) is a large-sized machine with 4 1.6GHz cores, 7GB RAM, 850GB of local storage, and 400Mbps allocated I/O bandwidth. Each instance was configured to support 5 map slots and 2 reduce slots. We experiment with up to 75 instances in our tests[4].

Datasets We use the following datasets in our evaluation, with dataset sizes based upon the aggregate amount of memory needed to run our queries over them:

- *Search data.* This is a real 100GB search dataset from Bing, that consists of userids and their search terms. The input splits were created by sharding the data into a number of files/partitions, and annotating with fine-grained PI values.

- *TPC-H data.* We used the *dbgen* tool to generate a 100GB TPC-H benchmark dataset, for experiments using TPC-H queries.

- *Click data.* This is a real 12GB dataset from the Microsoft AdCenter advertising platform, that comprises of clicks and impressions on various ads over a 3 month period.

Queries We use the following progressive queries:

- *Top-k correlated search.* The query reports the top-k words that are most correlated with an input search term, according to a *goodness score*, in the search dataset. The

[4]Our Windows Azure subscription allowed no more than 300 cores; this limited us to 75 4-core VM instances.

query consists of two NOW! jobs, one feeding the other. The first stage job uses the data set as input and partitions by userid. Each reducer computes a histogram that reports, for each word, the number of searches with and without the input term, and the total number of searches. The second stage job groups by word, and aggregates the histograms from the first stage, computes a per-word goodness, and performs top-k to report the k most correlated words to the input term. We use "music" as the default term.

- *TPC-H Q3.* We use a generalization of TPC-H query 3:

```
SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS
REVENUE, O_ORDERDATE, O_SHIPPRIORITY FROM ORDERS, LINEITEM
WHERE L_ORDERKEY = O_ORDERKEY
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
```

- *CTR.* The CTR (click-through-rate) query computes the MR job graph shown in Figure 4.3 (our running example). It consists of three queries (Q_c, Q_i, and Q_{ctr}) where Q_c is a *click query* which computes the number of clicks from the click dataset, Q_i is an *impressions query* which computes the number of ad impressions from the impression data set and Q_{ctr} computes the CTR.

Baselines We evaluate NOW! against several baseline systems:

- **Map-Reduce (MR).** For standard map-reduce, we use Daytona [80], a C# implementation of vanilla hadoop for Windows Azure. This baseline provides an estimate of time taken to process a partitioned query without progressive results.

- **Stateful MR (SMR).** Stateful MR is an extension of MR for iterative queries [81], that maintains reducer state across MR jobs. We use it for progressive results by chunking the input into batches, and submitting each chunk (in progress-sync order) as a separate MR job. Subsequent chunks use reducers that retain the prior job's state. For each chunk, we run each MR stage as a vanilla MR job. With multi-stage jobs, we process one chunk through all stages before submitting the next chunk to the first stage.

- **MRO [17].** MRO pipelines data between the mappers and reducers, but is unaware of progress semantics and does not use progress-sync merge at the reducers. This can lead to different nodes progressing at different speeds. We approximate MRO in NOW! by replacing the progress-aware merge with a union[5].

Job configuration and parameter settings. The configuration for a two-stage job (with one job feeding another) is depicted as $M_1 - R_1 - M_2 - R_2$ where M and R represent the number of mappers and reducers and their subscripts $(1, 2)$ represent the stage to which they belong (note that $R_1 = M_2$). A single stage job is depicted as $M_1 - R_1$. In our experiments, the number of mappers is equal to the number of input splits (stored as blobs). The number of reducers is chosen based on the memory capacity of each worker node (7GB RAM) and the number of mappers feeding the reducers.

[5]This baseline benefits from our other optimizations such as concurrent job scheduling, no sorting, and pipelining across stages.

4.5.3 Experiments and Results

We now present our evaluation results for NOW!.

Effect of Progressive Computation We evaluate NOW!'s performance vs. SMR in terms of time taken to produce progressive results. The first experiment (see Figure 4.10(a)) plots the time taken to run the top-k correlated search query which provides the top 100 words that were searched with "weather", in terms of progress-batches plotted in progress-sync order. The input data set was batched by the mapper into 75 progress-batches by NOW!. For the SMR baseline, the data was ordered and split into 75 chunks (one per PI). Each chunk representing one PI, was processed as a separate MR job and the time taken taken for the same was recorded. Each point on the plot represents an average of five runs. We used datasets of two sizes (6 and 8GB). The experimental results show that NOW! performs much better (6X improvement) than SMR, which processes each progress batch as a separate job and resorts to expensive intermediate output materialization, hurting performance, particularly in a Cloud setting. Also, the time taken for the first 50% of the progress batches is under 20mins as opposed to 105mins for SMR, for the 8GB dataset, highlighting the benefit of platform support for progressive early results.

Effect of Batching We evaluate the performance of NOW! for different progress-batch sizes and compare the same with SMR and MR. The MR baseline processes the entire input as a single batch. The granularity of batch size controls the number of progress batches. The dataset size used in this experiment is 6GB and the configuration is 94-26-26-4. The experiment shows the results for 3 different batch sizes: 80MB (75 batches),

600MB (10 batches) and 1200MB (5 batches), and compares them against vanilla MR which processes the entire input of 6GB at once.

Figure 4.10(b) shows the change in total query processing time with change in batch size. As the batch-size decreases from 1200MB to 80MB, the number of batches processed by the system increases from 5 to 75. The query processing time of SMR increases drastically with the increase in the number of batches, which can be attributed to the fact that it processes each batch as a separate MR job and resorts to intermediate data materialization. The MR baseline which processes the entire input as a single batch does better than SMR , but does not provide early results.

On the other hand, the query processing time for NOW! does not vary much with increase in number of batches as it is pipelined, does not start a new MR job for each batch, and does not materialize intermediate results between jobs. We do see a slight increase in query processing time when the number of batches increases from 10 to 75, which can be attributed to a moderate increase in batching overheads. However, the smallest batch-size provides the earliest progressive results and at the finest granularity. The figure shows the time to generate the first progress batch i.e., the time when the user starts getting progressive results. The time to first batch increases with increase in batch size (or sample size), but is significantly lower than the total query processing time.

Performance Breakdown We analyzed the performance of NOW! using our diagnostic monitoring module which logs CPU, memory, and I/O usage. Figure 4.10(c) analyses the performance of the two-stage top-k correlated search query with $k = 100$, and plots the % time taken by different components in NOW!. Each data point in the figure is an average

over 10 runs, for two different datasets (15GB and 30GB) on 30 machines. The results indicate that the maximum time is spent in the first stage reducer followed by the second stage reduce and writing the final output to the blobs. The framework does not have any major bottlenecks in terms of pipelining of progress-batches. The time taken by the two reduce stages would vary depending on the choice of progressive reducer and the type of query. Our current results use StreamInsight as the progressive reducer.

Scalability Figure 4.10(d) evaluates the effect of increase in data size on query processing time in NOW! as compared to SMR. We used the top-k correlated search query for the experiment and varied the data size from 2.8GB to 30GB. The results show that NOW! provides a scale-up of up to 38X over SMR in terms of reduction in query processing time. This can be attributed to pipelining, no sorting in the framework and no intermediate data materialization between jobs. Figure 4.10(e) shows the scale-up provided by NOW! in terms of throughput (#rows processed per second) with the increase in #machines. For the top-k correlated search query (top 100 words correlated to "music"), we achieved a 6X scale-up with 74 machines as compared to the throughput on 20 machines, for 15GB data.

Data Materialization Overheads Writing map outputs on the local disk, has a significant performance penalty, while on the other hand, intermediate data materialization provides higher availability in presence of failures . Figure 4.10(f) shows the overhead of disk I/O in materializing map output on disk and subsequent disk access to shuffle the data to the reducers within a job.

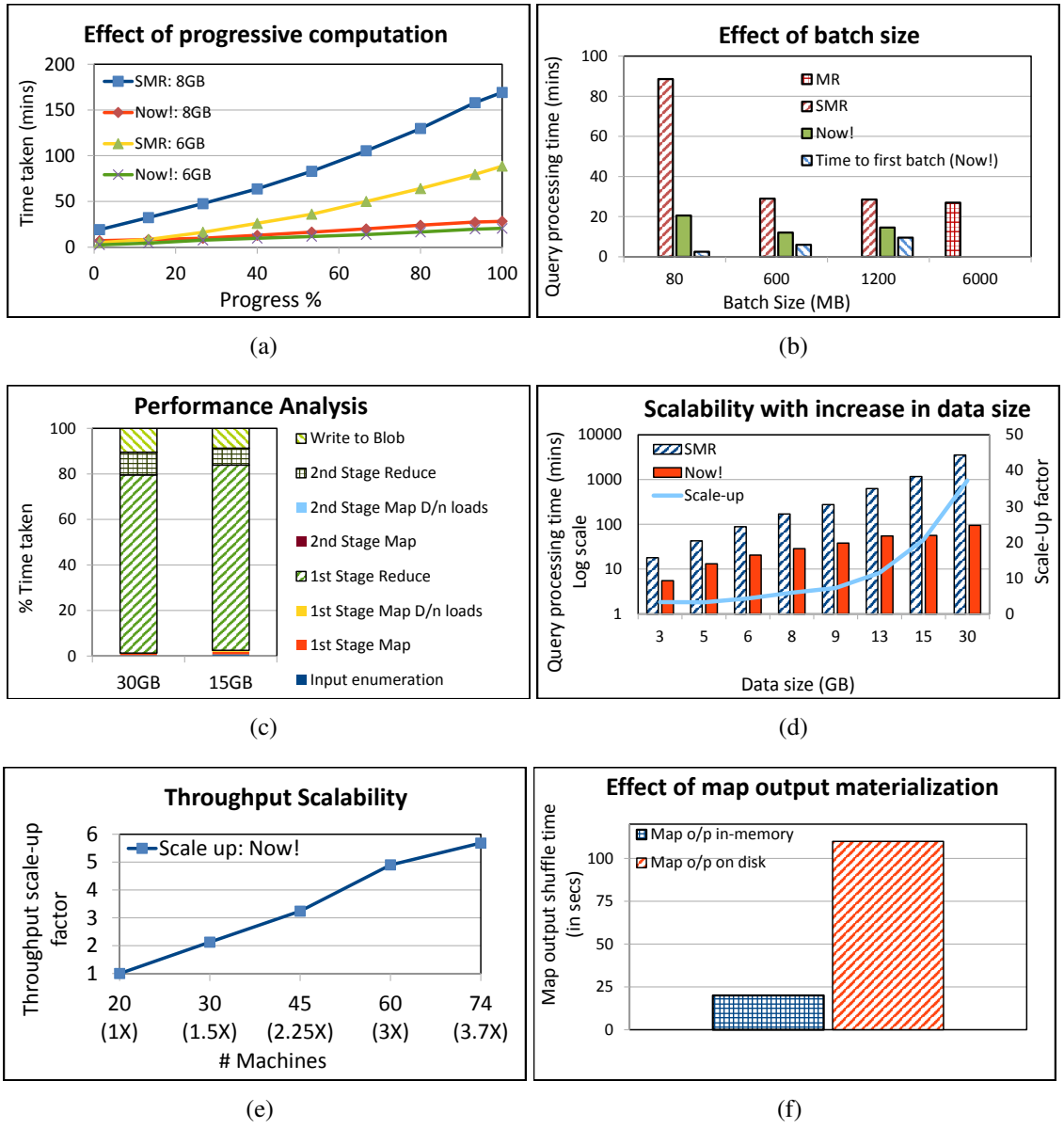


Figure 4.10: **Performance analysis.(a) Time taken to process a query in progress-sync order; (b) Effect of batching granularity; (c) Analysis of time taken by different elements for a two-stage Map-Reduce query. Scalability: (d) Effect of data size on query processing time; (e) Throughput scalability with increase in #machines; (f) Overheads of disk I/O (Map output materialization).**

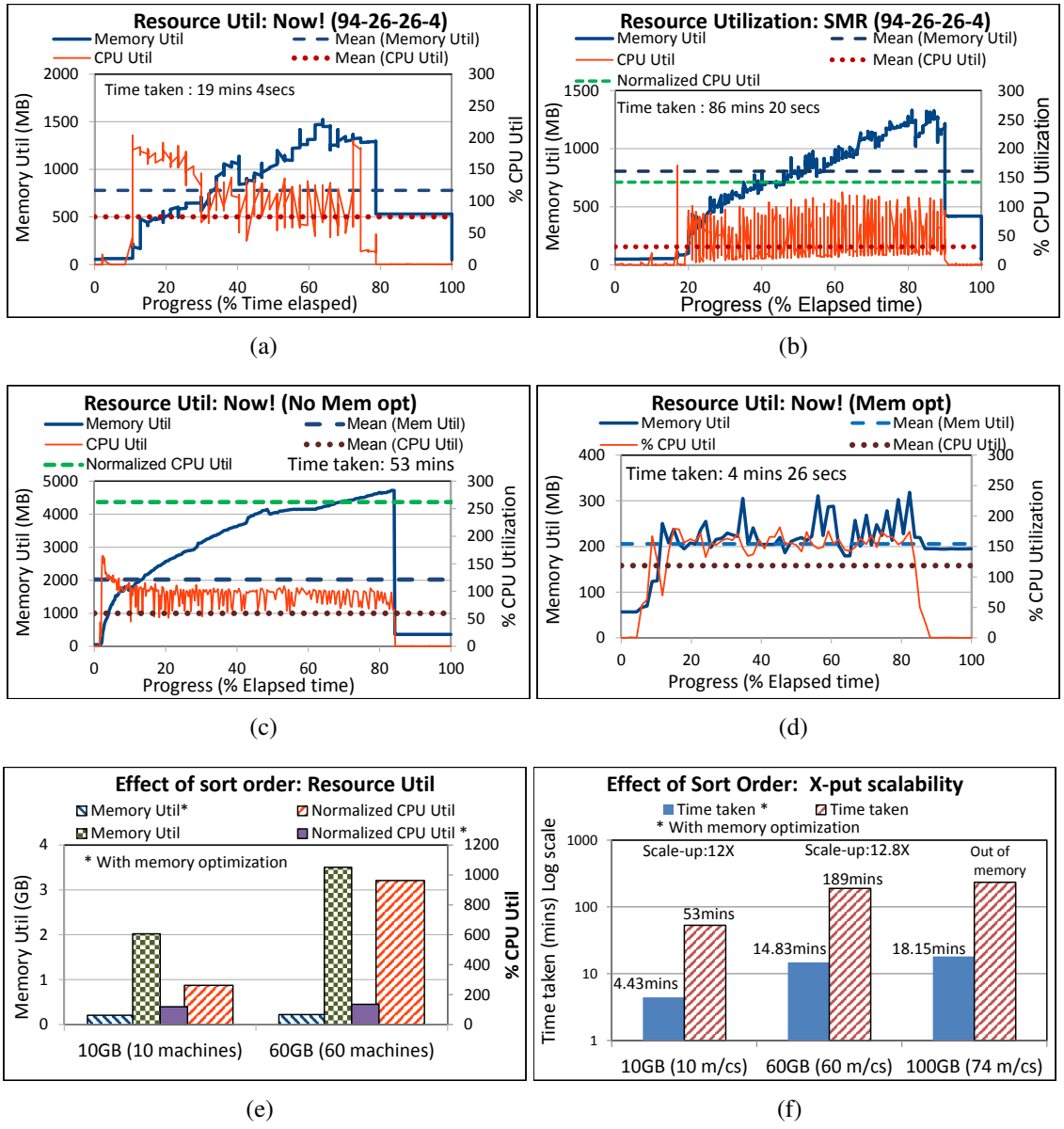


Figure 4.11: **Resource Utilization. (a) CPU and memory utilization NOW!; (b) CPU and memory utilization SMR;(c) CPU and memory utilization without memory optimization; (d) CPU and memory utilization with memory optimization. (e) Effect of sort order on memory and % CPU utilization for different data sizes; (f) Memory optimization effects on query processing time.**

Our results show an overhead of approx 90 secs for a dataset of 8GB for the 94-26-26-4 configuration. NOW! is tunable to work in both modes (with and without disk I/O) and can be chosen by the user depending on the application needs and the execution

environment. It is also pertinent to note here that there is no data materialization on persistent storage (HDFS or Cloud) between different Map-Reduce stages in NOW! which provides a similar performance advantage for multi-stage jobs over MR/SMR as seen in section 4.5.3.

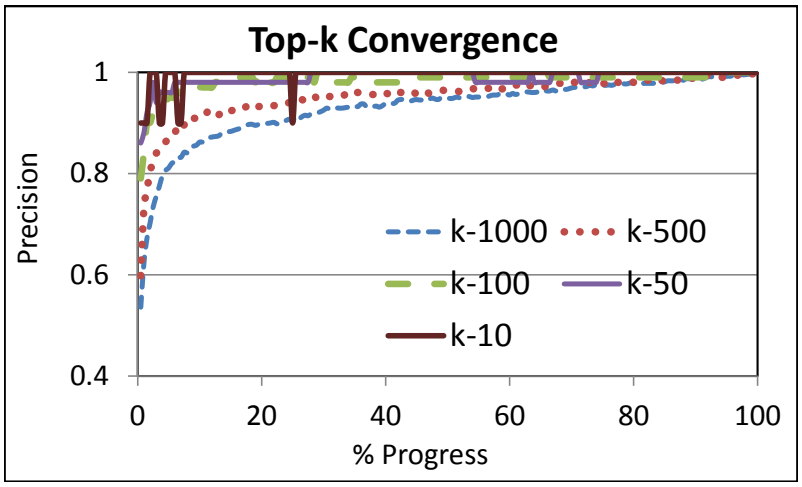
Resource Utilization We evaluated NOW! for its resource utilization in terms of memory and CPU. Figures 4.11(a,b) compare the memory and CPU utilization of NOW! and SMR for the 94-26-26-4 configuration for a dataset size of 8GB. The figures show the average real time memory and CPU utilization over 30 slave machines each running 4 mappers and 1 reducer plotted against time. The results indicate that there is no significant difference in the average memory utilization for both platforms, and the average CPU utilization of NOW! is actually higher than that of SMR. However, we also show the *normalized* %CPU utilization for SMR which is the product of the average CPU utilization and the *normalization factor* (ratio of time taken by SMR to the time taken by NOW!.) The normalized %CPU utilization is much higher as SMR takes approx 4.5X more time to complete as compared to NOW!. Thus, NOW! is ideal for progressive computation on the Cloud, where resources are charged by time.

Memory Optimization using Sort Orders The next experiment investigates the benefit of our memory optimization (cf. Section 4.2.7) in case the progress-sync order is correlated with the partitioning key. Our TPC-H dataset uses progress in terms of the L_ORDERKEY attribute, and TPC-H Q3 also partitions by the same key. An optimized run can detect this at compile-time and set $P^- = P^+ + 1$, allowing the query to "forget" previous tuples when we move to the next progress-batch. An unoptimized run would retain

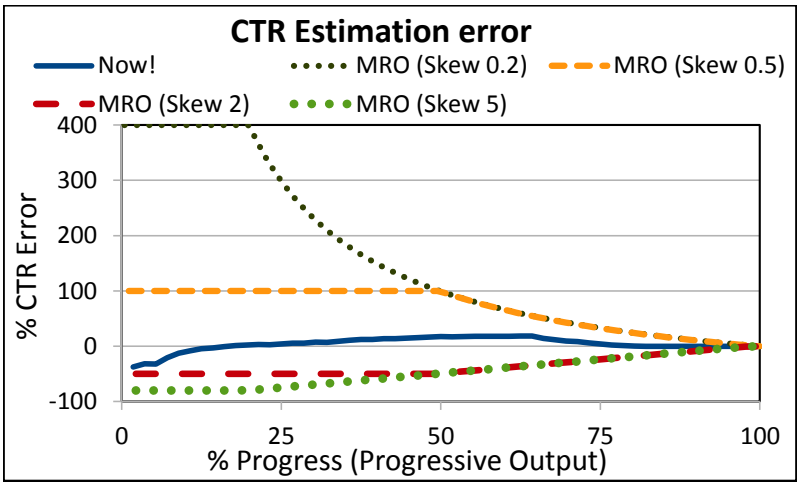
all tuples in memory in order to compute future join and aggregation results. We experiment with 10GB, 60GB and 100GB TPC-H datasets. Figures 4.11(c) and 4.11(d) show the variation of memory and CPU utilization with progress with and without memory optimization for the 10GB dataset. Figure 4.11(e) shows that the memory footprint of the optimized approach is much lower than the unoptimized approach, as expected. Further, it indicates that the lower memory utilization directly impacts CPU utilization since the query needs to maintain and lookup much smaller join synopses. Figure 4.11(f) shows that memory optimization gives an orders of magnitude reduction in time taken to process the TPC-H Q3 for all the three datasets providing a throughput scale-up of approx 12X in two cases (10GB and 60GB). As indicated in the figure, the 100GB run without memory optimization ran out of memory (OOM) as the data per machine was much higher.

Qualitative Evaluation: Result convergence In order to determine the speed of convergence we compute the precision (for the top-k correlated search query) of the progressive output values that we get as intermediate results. Figure 4.12(a) varies k and plots precision against the number of progress-batches processed for a data size of 15GB, with a configuration of 60-43-43-1 and 200 progress batches. The precision metric measures how close progressive results are to the final top-k. We see that precision quickly reaches 90%, after a progress of less than 20% as the top k values do not change much after sampling 20% of the data (lower k values converge quicker as expected). This shows the utility of early results for real-world queries where the results converge very quickly to the final answer after processing small amounts of data.

Qualitative Evaluation: Progress Semantics We compare result quality against an



(a)



(b)

Figure 4.12: **Qualitative analysis. (a) Top-k Convergence; (b) Error estimation of progressive results.**

MRO-style processing approach using the clicks dataset to compute CTR (Figure 4.3) progressively. We model variation in processing time using a *skew factor* that measures how much faster Q_i is, as compared to Q_c. A skew of 1 represents the hypothetical case where perfect CTR information is known a priori, and queries follow this relative processing speed. Figure 4.12(b) shows the % error in CTR estimation plotted against % progress. The experiment shows that if different queries proceed at different speeds, early results without user-defined progress semantics can become inaccurate (although

all techniques converge to the same final result). We see that even moderate skew values can result in significant inaccuracy. On the other hand, progress semantics ensure that the data being correlated always belongs to the same subset of users, which allows CTR to converge quickly and reliably, as expected.

4.6 Conclusion

Progressive data analytics can be used to as a means to reduce the cost of data analytics in the cloud. Data scientists typically perform progressive sampling to extract data for exploratory querying, which provides them user-control, determinism, repeatable semantics, and provenance. However, the lack of system support for such progressive analytics results in a tedious and error-prone workflow that precludes the reuse of work across samples. We designed and built a new system called NOW! based on PRISM that (1) allows users to communicate progressive samples to the system; (2) allows efficient and deterministic query processing over samples; and yet (3) provides repeatable semantics and provenance to data scientists. We showed that one can realize this model for *atemporal* relational queries using an unmodified *temporal* streaming engine, by re-interpreting temporal event fields to denote progress. NOW! has been built as a progressive data-parallel computation framework for the cloud, where progress is understood and propagated as a first-class citizen in the framework. NOW! works with StreamInsight to provide progressive SQL support over big data in Azure. Large-scale experiments showed orders-of-magnitude performance gains achieved by our solutions affecting substantial reduction in the cost of analytics in the cloud, without sacrificing the benefits offered by our underlying progress model.

Chapter 5: Neighborhood-centric Analytics on Large-scale Graphs in the Cloud

5.1 Introduction

There is an increasing interest in executing complex analyses over large graphs, many of which can be viewed as operations on local neighborhoods (or subgraphs of local neighborhoods) of a large number of nodes in the graph. For example, there is much interest in analyzing *ego networks*, i.e., 1- or 2-hop neighborhoods, of the nodes in the graph. Examples of specific ego network analysis tasks include identifying *structural holes*, brokerage analysis, counting *motifs* [82], identifying *social circles* (Figure 5.1) [50], link prediction and recommendations using Personalized Page Rank [83], computing *local clustering coefficients*, and anomaly detection [84]. In other cases, there may be interest in analyzing connected or induced subgraphs satisfying certain properties. As an example, we may be interested in analyzing the induced subgraph on users who tweet a particular *hashtag* in the Twitter network. Similarly, we may be interested in analyzing groups of users who have exhibited significant communication activity in recent past. More complex subgraphs can be specified as *unions* or *intersections* of neighborhoods of pairs of nodes; this may be required for graph cleaning tasks such as *link prediction* and *entity*

resolution.

Several vertex-centric distributed graph processing frameworks have been proposed in recent years, including Pregel [45], GraphLab [46], Apache Giraph [85], to name a few. In these frameworks, the users write vertex-level programs, that are then executed by the framework in either bulk synchronous or asynchronous fashion. The computation and execution models in these frameworks fundamentally limit the user program's access, to a single vertex's state (including its edges and in some cases neighbor IDs). This makes writing many of the above mentioned tasks in these frameworks not natural. For example, to analyze a 2-hop neighborhood to find social circles, one would first need to gather all the information from 2-hop neighbors through message-passing (a huge communication overhead), and reconstruct those neighborhoods locally (i.e., in the vertex program local state). This not only duplicates the graph processing functionality, but will likely be infeasible because high memory requirements arising from duplication of state. (Even reconstructing 1-hop neighborhoods locally, e.g., for counting triangles, increases the memory requirement by orders of magnitude). If the task is decomposable into smaller tasks, it may be possible to execute it in multiple steps through partial computations at a collection of nodes; however, most of these analysis tasks are not easily decomposable. Customizing these existing frameworks for analytics that require traversing beyond 1-hop neighbors may not be practical or efficient.

Most of these frameworks ignore the issues in extracting relevant portions of the underlying graph that an analysis task may be specifically interested in, and loading it onto distributed memory. In many cases, the user may only want to analyze a subgraph (or several subgraphs) of the overall graph, and may only need access to a subset of the node and

edge attributes.. Further, to minimize network communication due to distributed traversal during analysis, it is desired that vertices and edges be replicated so that the subgraphs that are analyzed by the user programs are entirely present in one of the partitions [86]. This leads to a critical challenge in the Cloud computing environment, that needs to be addressed to reduce the cost of analytics: given the user specification of an analysis task and neighborhoods of interest, how to load the relevant data onto a minimum number of partitions while minimizing the communication cost?

This dissertation introduces NSCALE, an end-to-end graph processing framework that enables scalable distributed execution of subgraph-centric analytics over large-scale graphs in the Cloud. In our framework, the user specifies: (a) the subgraphs of interest (for example, k-hop neighborhoods around vertices that satisfy a set of predicates) and (b) a user program to be executed on those subgraphs (which may itself be iterative). The user program is written against a general graph API (specifically, BluePrints), and has access to the entire state of the subgraph against which it is being executed. NSCALE execution engine is in charge of ensuring that the user program only has access to that state and nothing more; this guarantee allows existing graph algorithms to be used without modification. Thus a program written to compute, say, connected components in a graph, can be used as is to compute the connected components within each subgraph of interest.

Our current subgraph specification format allows users to specify subgraphs of interest as k-hop neighborhoods around a set of query vertices, followed by a filter on the nodes and the edges in the neighborhood. It also allows selecting subgraphs induced by certain attributes of the nodes; e.g., the user may choose an attribute like tweeted hashtags, and ask for induced subgraphs, one for each hashtag, over users that tweeted that

particular hashtag.

User programs corresponding to complex analytics may make arbitrary and random accesses to the graph they are operating upon. Hence, one of our key design decisions was to ensure that each of the subgraphs of interest would reside entirely in memory on a single machine while the user program ran against it. NSCALE consists of two major components. First, the graph extraction and packing (GEP) module extracts relevant subgraphs of interest and uses a cost-based optimizer for data replication and placement that minimizes the number of machines needed, while attempting to balance load across machines to guard against the straggler effect. Second, the distributed execution engine executes user-specified computation on the subgraphs in memory. It employs several optimizations that reduce the total memory footprint by exploiting overlap between subgraphs loaded on a machine, without compromising correctness.

Although we primarily focus on one-pass complex analysis tasks described above, NSCALE also supports the Bulk Synchronous Protocol (BSP) model for executing iterative analysis tasks like computation of PageRank or global connected components. NSCALE's BSP implementation is most similar to that of GraphLab, and the information exchange is achieved through shared state updates between subgraphs on the same partition and through use of "ghost" vertices (i.e., replicas) and message passing between subgraphs across different partitions.

We present a comprehensive experimental evaluation comparing against three state-of-the-art systems, namely, Giraph, GraphLab, and GraphX, on several real-world datasets and a variety of analysis tasks. Our results illustrates that extraction of relevant portions of data from the underlying graph and optimized data replication and placement helps

improve scalability and performance with significantly fewer resources reducing the cost of data analytics substantially. The graph computation and execution model employed by NSCALE affects a drastic reduction in communication (message passing) overheads (with no message passing within subgraphs), and significantly reduces the memory footprint (up to 2.6X for applications over 1-hop neighborhoods and up to 25X for applications such as personalized page rank over 2-hop neighborhoods); the overall performance improvements range from 3X to 30X for graphs of different sizes for applications over 1-hop neighborhoods and 20X to 400X for 2-hop neighborhood analytics. Further, our experiments show that GEP is a small fraction of the total time taken to complete the task, and is thus the crucial component that enables the efficient execution of the graph computation on the materialized subgraphs in distributed memory using minimal resources. This enables NSCALE to scale neighborhood-centric graph analytics to very large graphs for which the existing vertex-centric approaches fail completely.

NSCALE has been primarily deployed and tested on the Apache YARN platform. The details of the deployment architecture on YARN are discussed in Section 5.2. In addition to this, we have also implemented NSCALE on Apache Spark, the new big data management platform. We elaborate on the details of this implementation in Section 5.7.

5.2 NSCALE Overview

In this section we first discuss several representative graph analytics tasks that are ill-suited for vertex-centric frameworks, but fit well with NSCALE's subgraph-centric computation model. Subsequently we provide an overview of the programming model

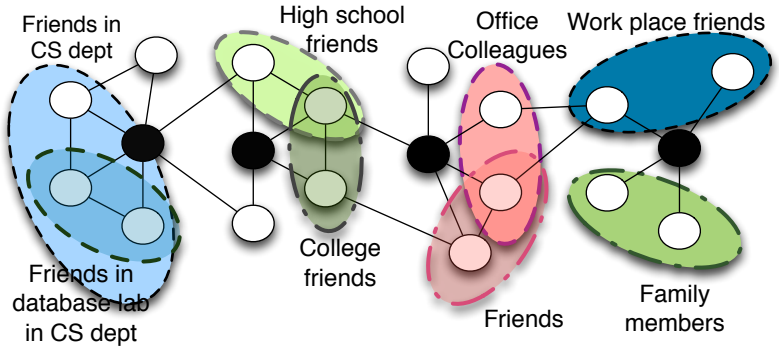


Figure 5.1: **An example of neighborhood-centric analysis: identify users' *social circles* in a social network.**

and architecture of NSCALE.

5.2.1 Application Scenarios

Local clustering coefficient (LCC). In a social network, the LCC quantifies, for a user, the fraction of his or her friends who are also friends—this is an important starting point for many graph analytics tasks. Computing the LCC for a vertex requires constructing its ego network, which includes the vertex, its 1-hop neighbors, and all the edges between the neighbors. Even for this simple task, the limitations of vertex-centric approaches are apparent, since they require multiple iterations to collect the ego-network before performing the LCC computation (such approaches quickly run out of memory as we increase the number of vertices we are interested in).

Identifying social circles. Given a user's social network (k-hop neighborhood), the goal is to identify the social circles (subsets of the user's friends), which provide the basis for information dissemination and other tasks. Current social networks either do this manually, which is time consuming, or group friends based on common attributes, which fails

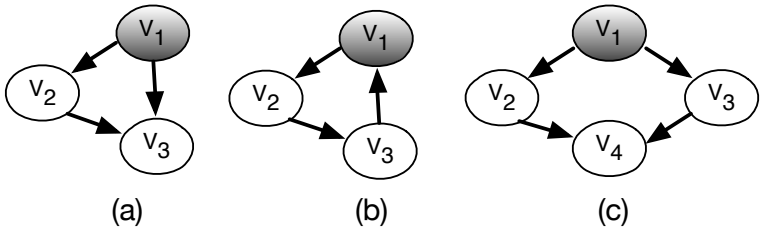


Figure 5.2: **Counting different types of network motifs: (a) Feed-fwd Loop, (b) Feedback Loop, (c) Bi-parallel Motif.**

to capture the individual aspects of the user's communities. Figure 5.1 shows examples of different social circles in the ego networks of a subset of the vertices (i.e., shaded vertices). Automatic identification of social circles can be formulated as a clustering problem in the user's k-hop neighborhood, for example, based on a set of densely connected alters [50]. Once again, vertex-centric approaches are not amenable to algorithms that consider subgraphs as primitives, both from the point of view of performance and ease of programming.

Counting network motifs. Network motifs are subgraphs that appear in complex networks (Figure 5.2), which have important applications in biological networks and other domains. However, counting network motifs over large graphs is quite challenging [87] as it involves identifying and counting subgraph patterns in the neighborhood of every query vertex that the user is interested in. Once again, in a vertex-centric framework, this would entail message passing to gather neighborhood data at each vertex, incurring huge messaging and memory overheads.

Social recommendations. Random walks with restarts (such as personalized PageRank [83]) lie at the core of several social recommendation algorithms. These algorithms can be implemented using Monte-Carlo methods [88] where the random walk starts at a vertex v,

and repeatedly chooses a random outgoing edge and updates a visit counter with the restriction that the walk jumps back only to v with a certain probability. The stationary distribution of such a walk assigns a PageRank score to each vertex in the neighborhood of v; these provide the basis for link prediction and recommendation algorithms. Implementing random walks in a vertex-centric framework would involve one iteration with message passing for each step of the random walk. In contrast, with NSCALE the complete state of the k-hop neighborhood around a vertex is available to the user's program, which can then directly execute personalized PageRank or any existing algorithm of choice.

Subgraph Pattern Matching and Isomorphism.

Subgraph pattern matching or subgraph isomorphism have important applications in a variety of application domains including biological networks, chemical interaction networks, social networks, and many others; and a wide variety of techniques have been developed for exact or approximate subgraph pattern matching [89–99] (see Lee et al. [100] for a recent comparison of the state-of-the-art techniques). Many of those techniques work by identifying potential matches for a central node in the pattern, and then exploring the neighborhood around those nodes to look for matches. This second step can often involve fairly sophisticated algorithms, especially if the patterns are large or contain sophisticated constructs, or if the goal is to find approximate matches, or if the data is uncertain. Most of those algorithms are not easily parallelizable, and hence it would not be easy to execute them in a distributed fashion using the vertex-centric programming frameworks. On the other hand, NSCALE could be used to construct the relevant neigh-

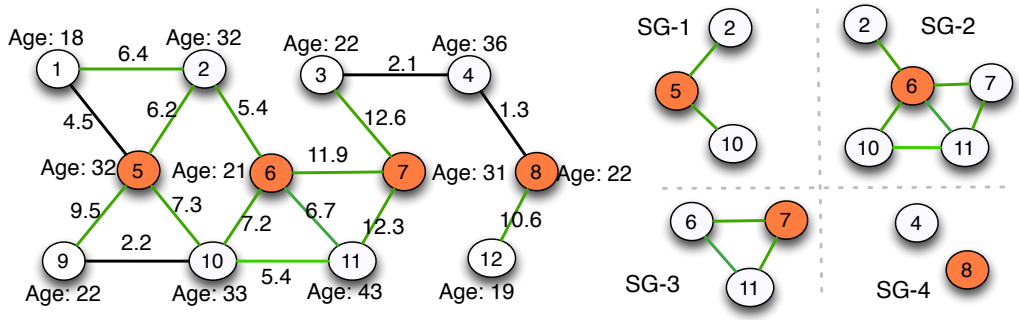


Figure 5.3: **A subgraph extraction query on a social network**

borhoods in memory in many of those cases, and those search algorithms could be used as is on those neighborhoods.

5.2.2 NSCALE Programming Model

We assume a standard definition of a graph $G(V, E)$ where $V = \{v_1, v_2, ..., v_n\}$ denotes the set of vertices and $E = \{e_1, e_2, ..., e_m\}$ denotes the set of edges in G. Let $A = \{a_1, a_2, ..., a_k\}$ denote the union of the sets of attributes associated with the vertices and edges in G. In contrast to vertex-centric programming models, NSCALE allows users to specify subgraphs or neighborhoods as the scope of computation. More specifically, users need to specify: (a) subgraphs of interest on which to run the computations through a *subgraph extraction query*, and (b) a user program.

Specifying subgraphs of interest. We envision that NSCALE will support a wide range of subgraph extraction queries, including pre-defined parameterized queries, and declaratively specified queries using a Datalog-based language that we are currently developing. Currently, we support extraction queries that are specified in terms of four parameters: (1)

```
ArrayList<RVertex> n_arr = new ArrayList<RVertex>();
for(Edge e: this.getQueryVertex().getOutEdges)
     n_arr.add(e.getVertex(Direction.IN));

int possibleLinks = n_arr.size()* (n_arr.size()-1)/2;

// compute #actual edges among the neighbors
for(int i=0; i < n_arr.size()-1; i++)
   for(int j=i+1; j < n_arr.size(); j++)
      if(edgeExists(n_arr.get(i), n_arr.get(j)))
          numEdges++;
double lcc = (double) numEdges/possibleLinks;
```

Figure 5.4: **Example user program to compute *local clustering coefficient* written using the BluePrints API. The *edgeExists()* call requires access to neighbors' states, and thus this program cannot be executed as is in a vertex-centric framework.**

a predicate on vertex attributes that identifies a set of *query vertices* (P_{QV}), (2) k – the radius of the subgraphs of interest, (3) edge and vertex predicates to select a subset of vertices and edges from those k-hop neighborhoods (P_E, P_V), and (4) a list of edge and vertex attributes that are of interest (A_E, A_V). This captures a large number of subgraph-centric graph analysis tasks, including all of the tasks discussed earlier. For a given subgraph extraction query q, we denote the subgraphs of interest by $SG_1(V_1, E_1), ..., SG_q(V_q, E_q)$.

Figure 5.3 shows an example subgraph extraction query, where the query vertices are selected to be vertices with $age > 18$, radius is set to 1, and the user is interested in extracting induced subgraphs containing vertices with $age > 25$ and edges with $weight > 5$. The four extracted subgraphs, $SG_1, ..., SG_4$ are also shown.

Specifying subgraph computation user program. The user computation to be run against the subgraphs is specified as a Java program against the BluePrints API [101], a collection of interfaces analogous to JDBC but for graph data. Blueprints is a generic graph Java API used by many graph processing and programming frameworks (e.g., Gremlin,

a graph traversal language [102]; Furnace, a graph algorithms package [103]; etc.). By supporting the Blueprints API, we immediately enable use of many of these already existing toolkits over large graphs. Figure 5.4 shows a sample code snippet of how a user can write a simple local clustering coefficient computation using the BluePrints API. The subgraphs of interest here are the 1-hop neighborhoods of all vertices (by definition, a 1-hop neighborhood includes the edges between the neighbors of the node).

NSCALE supports the **Bulk Synchronous Protocol (BSP) for iterative execution**, where the analysis task is executed using a number of iterations (also called *supersteps*). In each iteration, the user program is independently executed in parallel on all the subgraphs (in a distributed fashion). The user program may then change the state of the query vertex on which it is operating (for consistent and deterministic semantics, we only allow the user program to change state of the query vertex that it owns; otherwise we would need a mechanism to arbitrate conflicting changes to a vertex state and we are not aware of any clean and easy model for achieving that). The state changes are made visible across all the subgraphs during the synchronization barrier, through use of shared state for subgraphs on the same partition and through message passing for subgraphs on different partitions. We provide a more detailed description of the provision of support for iterative computation in NSCALE, including the consistency and ownership model used, in Section 5.4.3.

Certain user applications might require customized aggregation of the values produced as a result of executing the user-specified program on the subgraphs of interest. Our mechanism to handle state updates for iterative tasks can also be used for aggregating information across all the nodes in the graph in the synchronization step. To briefly

summarize, the nodes can send messages to the coordinator that it can use to make various decisions (e.g., when to stop). The messages can be first locally aggregated, and the final aggregation is done by the coordinator (depending on the aggregation function).

5.2.3 System Architecture

Figure 5.5 shows the overall system architecture of NSCALE, which is implemented as a Hadoop YARN application. The framework supports ingestion of the underlying graph in a variety of different formats including edge lists, adjacency lists, and in a variety of different types of persistent storage engines including key–value pairs, specialized indexes stored in flat files, relational databases, etc. The two major components of NSCALE are the graph extraction and packing (GEP) module and the distributed execution engine. We briefly discuss the key functionalities of these two components here, and present details in the following sections.

Graph Extraction and Packing (GEP) Module. The user specifies the subgraphs of interest and the graph computation to be executed on them using the NSCALE user API. Unlike prior graph processing frameworks, the GEP module forms a major component of the overall NSCALE framework. From a usability perspective, it is important to provide the ability to read the underlying graph from the persistent storage engines that are not naturally graph-oriented. However, more importantly, partitioning and replication of the graph data are more critical for graph analytics than for analytics on, say, relational or text data.

Graph analytics tasks, by their very nature, tend to traverse graphs in an arbitrary

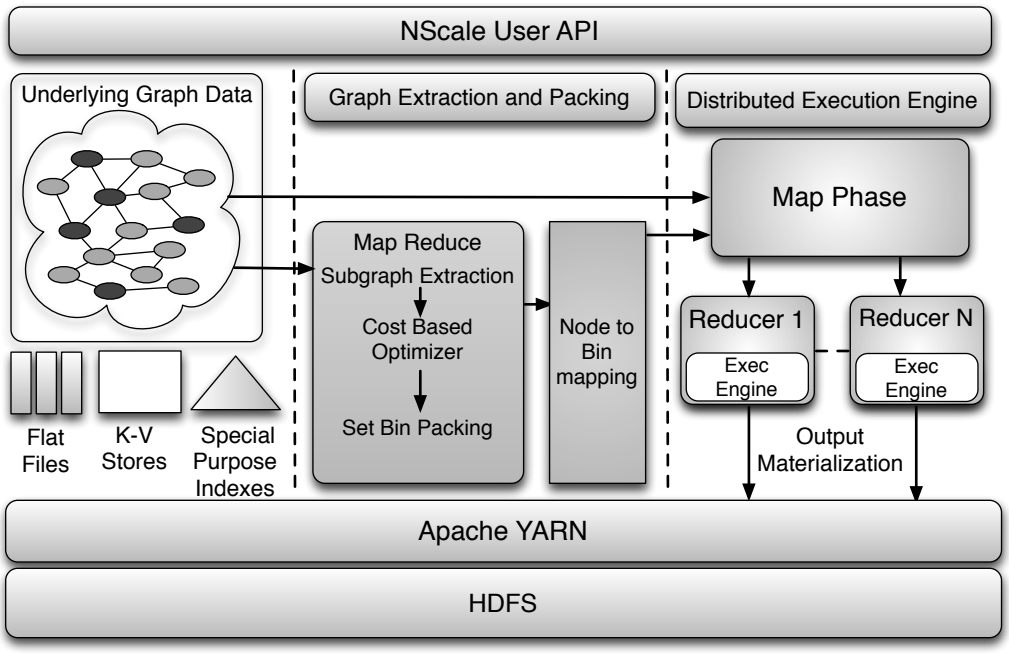


Figure 5.5: **NSCALE architecture. The GEP module is responsible for extracting and packing subgraphs of interest and then handing off the partitions to the distributed execution engine.**

and unpredictable manner. If the graph is partitioned across a set of machines, then many of these traversals are made over the network, incurring significant performance penalties. Further, as the number of partitions of a graph grows, the number of *cut* edges (with endpoints in different partitions), and hence the number of distributed traversals, grows in a non-linear fashion. This is in contrast to relational or text analytics where the number of machines used has a minor impact on the execution cost.

This is especially an issue in NSCALE, where user programs are treated as black-boxes. Hence, we have made a design decision to avoid distributed traversals altogether by replicating vertices and edges sufficiently so that every subgraph of interest is fully present in at least one partition. Similar approach has been taken by some of the prior work on efficiently executing "fetch neighbors" queries [104] and SPARQL queries [86]

in distributed settings. The GEP module is used to ensure this property, and is responsible for extracting the subgraphs of interest and packing them onto a small set of partitions such that every subgraph of interest is fully contained within at least one partition. GEP is implemented as multiple MapReduce jobs (described in detail later). The output is a *vertex-to-partition mapping*, which consists of a mapping from the graph vertices to partitions to be created. This data is either written to HDFS or directly fed to the execution engine.

Distributed Execution Engine. The distributed execution phase in NSCALE is implemented as a MapReduce job, which reads the original graph and the mappings generated by GEP, shuffles graph data onto a set of reducers, each of which constructs one of the partitions. Inside each reducer, the execution engine is instantiated along with the user program, which then receives and processes the graph partition.

The execution engine supports both serial and parallel execution modes for executing user programs on the extracted subgraphs. For serial execution, the execution engine uses a single thread and loops across all the subgraphs in a partition, whereas for parallel execution, it uses a pool of threads to execute the user computation in parallel on multiple subgraphs in the partition. However, this is not straightforward because the different subgraphs of interest in a partition are stored in an overlapping fashion in memory to reduce the total memory requirements. The execution engine employs several bitmap-based techniques to ensure correctness in that scenario.

5.3 Graph Extraction and Packing

5.3.1 Subgraph Extraction

Subgraph extraction in the GEP module has been implemented as a set of MapReduce (MR) jobs. The number of MR stages needed depends on the size of the graph, how the graph is laid out, size(s) of the machine(s) available to do the extraction, and the complexity of the subgraph extraction query itself. The first stage of GEP is always a map stage that reads in the underlying graph data, and identifies the *query vertices*. It also applies the filtering predicates (P_E, P_V) to remove the vertices and edges that do not pass the predicates. It also computes a *size* or *weight* for each vertex, that indicates how much memory is needed to hold the vertex, its edges, and their attributes in a partition. This allows us to estimate the memory required by a subgraph as the sum of the weights of its constituent vertices. (Only the attributes identified in the extraction query are used to compute these weights.) The rest of the GEP process only operates upon the network structure (the vertices and the edges), and the vertex weights.

Case 1: Filtered graph structure is small enough to fit in a single machine. In that case, the vertices, their weights, and their edges are sent to a single reducer. That reducer constructs the subgraphs of interest and represents them as subsets of vertices, i.e., each subgraph is represented as a list of vertices along with their weights (no edge information is retained further); this is sufficient for the subgraph packing purposes. The subgraph packing algorithm takes as input these subsets of vertices and the vertex weights, and produces a vertex-to-partition mapping.

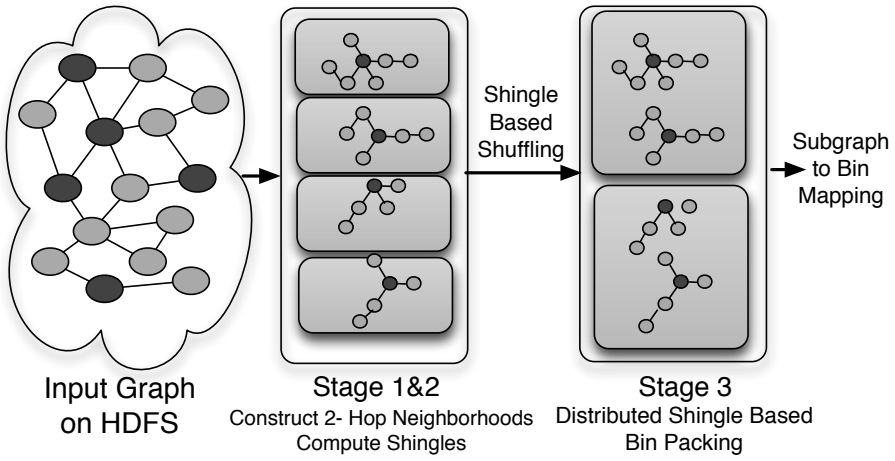


Figure 5.6: **Distributed GEP Architecture: Stages 1 and 2 construct the 2-hop neighborhoods; Stage 3 does the distributed shingle based bin packing producing the final subgraph to bin mapping.**

Case 2: Filtered graph structure does not fit on a single machine. In that case, the subgraph extraction and packing both are done in a distributed fashion, with the number of stages dependent on the radius (k) of subgraphs of interest.

We explain the process assuming $k = 2$, i.e., assuming our subgraphs of interest are 2-hop neighborhoods around a set of query vertices. We also assume an adjacency list representation of the data[1] (i.e., the IDs of the neighbors of a vertex are stored along with rest of its attributes);

Figure 5.6 shows the 3-stage distributed architecture of GEP. We begin with providing a brief sketch of the process. Given an input graph and a user query, the first two stages essentially are responsible for gathering for each query-vertex, its 2 hop neighborhood along with the weight attributes associated with each vertex in the 2-hop neighborhood. This is done iteratively, wherein the first stage constructs the 1-hop neighborhood

[1]For input graphs represented as an edge list with the vertex attributes available as a separate mapping, we have a minor modification to the first stage that uses a MapReduce job to join the edge and vertex data and produce a distributed adjacency list in the required format.

of the query-vertices specified by the query with all the required information on a set of reducers. Subsequently, the second stage takes the output of the first stage as input, constructs the 2-hop neighborhoods of the query-vertices and computes their shingle values in a distributed fashion, and outputs them as keys associated with these query-vertex neighborhoods. The final stage shuffles the neighborhoods based on these keys to multiple reducers in an attempt to group together neighborhoods with high overlap on a single reducer. The reducers in stage 3 run the bin packing in parallel which is followed by a post-processing step to produce the final neighborhood-to-bin mapping.

Next, we provide an in-depth description of the process. For a node u, let $N(u) = u_1, ..., u_{N(u)}$ denote its neighbors. The following steps are taken:

MapReduce Stage 1: For each vertex u that passes the filtering predicates (P_V), the map stage emits $N(u) + 1$ records:

$\langle key, (u, weight(u), isQueryVertex, N(u))\rangle$,

where $key = u, u_1, ..., u_{N(u)}$. Thus, given a vertex u, we have $N(u)' + 1$ records that were emitted with u as the key, one for its own information, and one for each of its $N(u)'$ neighbors that satisfies P_V (emitted while those neighbors are processed). In the reduce stage, the reducer responsible for vertex u now has all the information for its 1-hop neighbors, and IDs of all its 2-hop neighbors (obtained from its neighbors' neighborhoods), but it does not have the weights of its 2-hop neighbors or whether they satisfied the filtering predicates P_V. For each query vertex u, the reducer creates a list of the nodes in its 2-hop neighborhood, and outputs that information with key u. For each vertex v and for each of its 2-hop neighbors w, it also emits a record $\langle key = w, (v, weight(v))\rangle$.

MapReduce Stage 2: The second MapReduce stage groups the outputs of the first MapReduce stage by the vertex ID. Each reducer processes a subset of the vertices. There are two types of records that a reducer might process for a vertex u: (a) a record containing a list of u's 1- and 2-hop neighbors and the weights of its 1-hop neighbors, and (b) several records each containing the weight of a 2-hop neighbor of u. If a reducer only sees the records of the second type, then u is not a query vertex, and those records are discarded. Otherwise, the reducer adds the weight information for 2-hop neighbors, and completes the subgraph corresponding to u. For each of the subgraphs, the reducer then computes a min-hash signature, i.e., a set of *shingles*, over the vertex set of the subgraph, and emits a record with the set of shingles as the key and the subgraph as the value (we use 4 shingles in our experiments). A shingle is computed by applying a hash function to each of the vertex IDs in the subgraph, and taking the minimum of the hash values; it is well known that if two sets share a large fraction of the shingles, then they are likely to have a high overlap [105].

MapReduce Stage 3: The third MapReduce phase uses the shingle value of the subgraphs to shuffle the subgraphs to appropriate reducers. As a result of this shuffling, the subgraphs that are assigned to a reducer are likely to have high overlap and the subgraph packing algorithm is executed on each reducer separately. Finally, a post-processing step combines the results of all the reducers by merging any partitions that might be underutilized in the solutions produced by the individual reducers.

Intuitively, the above sequence of MapReduce stages constructs the required subgraphs, and then does a shuffle using the shingles technique in an attempt to create groups

that contain overlapping subgraphs. Those groups are then processed independently and the resulting vertex-to-partition mappings are concatenated together.

5.3.2 Subgraph Packing

Problem Definition. We now formally define the problem of packing the extracted subgraphs into a minimum number of partitions (or bins)[2], such that each subgraph is contained within a partition and the computation load across the partitions is balanced. Let $SG = \{SG_1, SG_2, .., SG_q\}$ be the set of subgraphs extracted from the underlying graph data (at a reducer). As discussed earlier, we assume that the memory required to hold a subgraph SG_i can be estimated as the sum of weights of the nodes in it. Let BC denote the bin capacity. This is set based on the maximum container capability of a YARN cluster node, a configuration parameter that needs to be set for the YARN cluster keeping in mind the maximum allocation of resources to individual tasks on the cluster.

Without considering overlaps between subgraphs and the load balancing objective, this problem reduces to the standard *bin packing* problem, where the goal is to minimize the number of bins required to pack a given set of objects. The variation of the problem where the objects are *sets*, and when packing multiple such objects into a bin, a *set union* is taken (i.e., overlaps are exploited), has been called *set bin packing*; that problem is considered much harder and we have found very little prior work on that problem [106].

Further, we note that we have a dual-objective optimization problem; we reduce it to a single-objective optimization problem by putting a constraint on the number of subgraphs that can be assigned to a bin. Let MAX denote the constraint, i.e., the maximum

[2]We use the terms *partitions* and *bins* interchangeably in this chapter.

number of subgraphs that can be assigned to a bin.

Subgraph Bin Packing Algorithms. The subgraph bin packing problem is NP-Hard and appears to be much harder to solve than the standard *bin packing* problem, as it also exhibits some of the features of the *set cover* and the *graph partitioning* problems. Next, we develop several scalable heuristics to solve this problem. We also developed and implemented an optimal algorithm for this problem (OPT), where we construct an Integer Program for the given problem instance and use the Gurobi Optimizer to solve the Integer Program. We were, however, able to run OPT successfully only for a very few small graphs; we present those results in Section 5.6.2.

5.3.2.1 Bin Packing-based Algorithms

The first set of heuristics that we develop exploit the similarity between subgraph packing problem and the bin packing problem. All of these heuristics use the standard greedy bin packing algorithm, where the items are considered in a particular order and placed in the first bin where they fit. More specifically, the algorithm (Algorithm 5) takes as input an ordered list of subgraphs, as determined by the heuristic, processes them in order, and packs each subgraph into the first available bin that has the available residual capacity, without violating the constraint on the maximum number of subgraphs in a bin. The addition of a subgraph to a bin is a set union operation that takes care of the overlap between the subgraphs. Each bin represents a partition onto which the actual graph data, associated with the nodes mapped to the bin using this algorithm, would be distributed for final execution step.

The complexity of this algorithm in the worst case in terms of the number of com-

Algorithm 5: Bin Packing Algorithm.

Input : Ordered list of subgraphs $SG_1, ..., SG_q$, each represented as a list of vertices and edges
Input : Bin capacity BC; Maximum number of subgraphs per bin MAX
Output: Partitions

```
for i = 1, 2, ..., q do
    for j = 1, 2, ..., B do
        if number of subgraphs in Bin j < MAX then
            if SG_i fits in Bin j (accounting for overlap) then
                Add SG_i to Bin j;
                break;
            end
        end
    end
    if SG_i not yet placed in a bin then
        Create a new bin and add SG_i to it;
    end
end
```

parison operations required is $\mathcal{O}(nm)$ where n is the number of subgraphs and m is the number of bins required ($= n$ in the worst case). Each comparison operation compares the estimated size of the union (accounting for the overlap) and the bin capacity. In addition to these comparisons, there would be n set union operations for inserting the subgraphs into bins. The complexity of the comparison and the set union operations is implementation dependent. For a hashtable-based approach, those operations would be linear in the number of set elements, giving us an overall complexity of $\mathcal{O}(nmC)$, where C is the bin capacity. However this worst-case complexity is quite pessimistic, and in practice, the algorithms run very fast.

We now describe three different heuristics to provide the input ordering of the subgraphs to be packed into bins.

1. First Fit bin packing algorithm. The first fit algorithm is a standard greedy 2-

approximation algorithm for bin packing, and processes the subgraphs in the order in which they were received (i.e., in arbitrary order).

2. First Fit Decreasing bin packing algorithm. The first fit decreasing algorithm is a variant of the first fit algorithm wherein the subgraphs are considered in the decreasing order of their sizes.

3. Shingle-based bin packing algorithm. The key idea behind this heuristic is to order the subgraphs with respect to the similarity of their vertex sets. The ordering so produced will maximize the probability that subgraphs with high overlap are processed together, potentially resulting in a better overall packing.

The shingle-based ordering is based on the min-hashing technique [107] which produces signatures for large sets that can be used to estimate the similarity of the sets. For computing the *min-hash* signatures (or shingles) of the subgraphs of interest over their vertex set, we choose a set of k different random hash functions to simulate the effect of choosing k random permutations of the characteristic matrix that represents the subgraphs. For each query vertex and each hash function, we apply the hash function to the set of nodes in the subgraph of the query vertex and find the minimum among the hash values.

Thus the output of the shingle computation algorithm (Ref Algorithm 6) is a list of k shingles (min-hash values) for each subgraph of interest, where the order of the hash functions within the list is effectively arbitrary[3]. To compute the shingle ordering, we sort-order the subgraphs of interest based on this list of shingle values associated with the

[3]The higher the value of k, the better the quality of the result. We have chosen $k = 6$ for our implementation which was determined experimentally to strike a fine balance between the quality of shingle-based similarity and computation time.

Algorithm 6: Computing shingles for a subgraph

Input : Subgraph $SG(V, E)$; A family of pairwise-independent hash functions H
$shingles[SG_i] \leftarrow \{\}$;
for $h \in H$ **do**
  $shingles[SG] \leftarrow \{shingles[SG], min_{v \in V} h(v)\}$;
end
return shingles;

subgraphs in a lexicographical fashion. The sorted order so obtained using this technique places subgraphs with high Jaccard similarity (i.e., overlap) in close proximity to each other. This shingle-based order is then used to pack the neighborhoods into bins using the greedy algorithm.

Handling skew. A high variance in the sizes of subgraphs could lead to a bin packing where some partitions have only a few large subgraphs and few partitions have a very large number of small subgraphs. This might lead to load imbalance and skewed execution times across partitions. To handle this skew in the sizes of the subgraphs, the bin packing algorithm (Algorithm 1) accepts a constraint on the maximum number of subgraphs (MAX) in a bin in addition to the bin capacity. This limits the number of small subgraphs that can be binned together in a partition and mitigates the potential of load imbalance between partitions to some degree. The trade-off here is that, we may need to use a higher number of bins to satisfy the constraints while some of the bins are not fully utilized. The MAX parameter can be set empirically depending on the nature of user computation and the underlying graph keeping in view the above mentioned trade-off.

5.3.2.2 Graph Partitioning-based Algorithms

The subgraph packing problem has some similarities to the *graph partitioning* problem, with the key difference being that: standard graph partitioning problem asks for disjoint balanced partitions, whereas the partitions that we need to create typically have overlap in order to satisfy the requirement that each subgraph be completely contained within at least one partition. Graph partitioning is very well-studied and a number of packages are available that can partition large graphs efficiently, METIS perhaps being the most widely used [108].

Despite the similarities, graph partitioning algorithms turn out to be a bad fit for the subgraph packing problem, because it is not easy to enforce the constraint that each subgraph of interest be completely contained in a partition. One option is to start with a disjoint partitioning returned by a graph partitioning algorithm, and then "grow" each of the partitions to ensure that constraint. However, we also need to ensure that the enlarged partitions obey the bin capacity constraint, which is hard to achieve since different partitions may get enlarged by different amounts.

We instead take the following approach (Algorithm 7). We *over partition* the graph using a standard graph partitioning algorithm (we use METIS in our implementation) into a large number of fine-grained partitions. We then grow each of those partitions as needed. This requires that for each query vertex in the fine grained partition, we check is its k-hop neighborhood lies within the partition. If not, we replicate the required nodes in the partition. This ensures that each subgraph of interest is fully contained in one of the partitions, and finally use the shingle-based bin packing heuristic to pack those

Algorithm 7: Graph Partitioning-based algorithm.

Input : Graph $G(V, E)$; Num of over partitions k
Output: Bins $\mathcal{B}$
//Over partition G into k partitions.;
$\mathcal{P} \leftarrow$ Metis(G); where $|\mathcal{P}| = k$;
for $p \in P$ **do**
  for $qv \in p$ **do**
    if *!* $(k - hop\ neighborhood) \in p$ **then**
      Grow: Replicate the required nodes adding them to p;
    end
  end
end
//Compute Shingles for each grown partition;
for *i = 1 to* $|P|$ **do**
  $s_i = ComputeShingles(p_i)$;
end
//Sort the partitions based on shingle values (s_i) ;
Sort(P);
$\mathcal{B} = BinPackingAlgo(P)$;
return $\mathcal{B}$;

partitions into bins. While packing, we also keep track of the nodes that are owned by the bin (or partition) and the ones that are replicated (ghosts) from other bins, to maintain the invariant of keeping each subgraph of interest fully in the memory of one of the partitions.

5.3.2.3 Clustering-based Algorithms

The subgraph packing problem also has similarities to *clustering*, since our goal can be seen as identifying similar (i.e., overlapping) subgraphs and grouping them together into bins. We developed two heuristics based on the two commonly used clustering techniques.

Agglomerative Clustering-based Algorithm. Agglomerative clustering refers to a class

of bottom-up algorithms that start with each item being in its own cluster, and recursively merge the closest clusters till the requisite number of clusters is reached. For handling large volumes of data, a threshold-based approach is typically used where in each step, pairs of clusters that are sufficiently close to each other are merged, and the threshold is slowly increased. Next we sketch our adaptation of this technique to subgraph packing.

We start with computing a set of shingles for each subgraph and ordering the subgraphs in the shingle order. This is done in order to reduce the number of pairs of clusters that we consider for merging; in other words, we only consider those pairs for merging that are sufficiently close to each other in the shingle order. The function $createAggClusters()$ in Algorithm 8 does the actual scanning of sets and merges close by sets together. The algorithm uses two parameters, both of which are adapted during the execution: (1) τ, a threshold that controls when we merge clusters, and (2) l, that controls how many pairs of clusters we consider for merging. In other words, we only merge a pair of clusters if they are less than l apart in the shingle order, and the Jaccard distance between them is less than τ. The set of merged clusters are available as AC.

To reduce the number of parameters, we use a sampling-based approach in the function $setThreshold()$ in Algorithm 8, to set τ at the beginning of each iteration. We choose a random sample of the eligible pairs (we use 1% sample), compute the Jaccard distance for each pair, and set τ such that 10% of those pairs of clusters would have distances below τ. We experimented with different percentage thresholds, and we observed that 10% gave us the best mix of quality and running time.

After computing τ, we make a linear scan over the clusters that have been constructed so far. For each cluster, we compute its actual Jaccard distance with the l clusters

Algorithm 8: Agglomerative Clustering-based algorithm.

Input : Set of subgraphs $SG = \{SG_1, ..., SG_q\}$
Input : Merge size l (Number of pairs to be considered for merging.)
Output: Agglomerative Clusters (Bins) $\mathcal{AC}$
//Compute Shingles of each subgraph;
for $i = 1$ *to* q **do**
    $s_i = ComputeShingles(SG_i)$;
end
/*Sort the subgraphs based on their shingle values ($S = \{s_1, s_2, ..s_q\}$)*/;
Sort(SG) ;
Done=false;
//Create an empty set of agglomerative clusters;
$\mathcal{AC} \leftarrow \phi$;
while *!Done* **do**
    τ = setThreshold();
    $numMerges = createAggCluster(SG, \mathcal{AC}, \tau, I)$;
    if $numMerges = 0$ **then**
        Done=True;
        break;
    end
    //adjust the merge size if required;
    I = adjustMergeSize();
    //Re-Compute Shingles of each merged cluster;
    $m = |\mathcal{AC}|$;
    for $i = 1$ *to* m **do**
        $s_i = ComputeShingles(AC_i)$;
    end
    //Sort clusters based on their shingle values (s_i). Sort(AC) ;
    $SG = \mathcal{AC}$;
end
return $\mathcal{AC}$;

that follow it. If the smallest of those distances is less than τ, then we merge the two clusters and re-compute shingles for the merged cluster (this is done by simply picking the minimum of the two values for each shingle position). This is only done if the merged cluster does not exceed the bin capacity (pairs of clusters whose union exceeds bin capacity are also excluded from the computation of τ).

During computation of τ, we also keep track of the number of pairs excluded because the size of their union is larger than the bin capacity. If those pairs form more 50% of sampled pairs, then we increase l ($adjustMergeSize()$) to increase the pool of eligible pairs. Since this usually happens towards the end when the number of clusters is small, we do this aggressively by increasing l by 50% each time. The algorithm halts when it cannot merge any pair of clusters without violating the bin capacity constraint.

K-Means-based Algorithm. K-Means is perhaps the most commonly used algorithm for clustering, and is known for its scalability and for constructing good quality clusters. Our adaptation of K-means (Ref Algorithm 9) is sketched next.

We start by picking k of the subgraphs randomly as *centroids*. We then make a linear scan over the subgraphs and for each subgraph, we compute the distance to each centroid using the function $computeDistance()$. We assign the subgraph to the centroid with which it has the highest intersection (in other words, we assign it to the centroid whose size needs to increase the least to include the subgraph). This is only done if the total size of the vertices in the cluster does not exceed BC. After assigning the subgraph to the centroid, we recompute the centroid ($UpdateCentroid()$) as the union of the old centroid and the subgraph. The function also keeps track of multiplicities of the vertices

Algorithm 9: KMeans Clustering-based algorithm.

Input : Set of subgraphs $SG = \{SG_1, ..., SG_q\}$; Bin Capacity BC
Input : k: The number of K-Means Clusters; MAX: maximum iterations
Output: Bins $\mathcal{B}$
//Create an empty centroid set $\mathcal{KC} \leftarrow \phi$;
//Randomly pick k subgraphs and assign them as the k-centroids;
while *(Sizeof($\mathcal{KC}$) $< k$)* **do**
    //Generate a random number from 1 to k i=GenerateRandom(k);
    $\mathcal{KC} = \mathcal{KC} \bigcup SG_i$
end
//Scan over the set of subgraphs and assign them to nearest centroid;
$AssignmentMap \leftarrow \phi$;
for *i = 1 to q* **do**
    if *!($SG_i \in \mathcal{KC}$)* **then**
        Max = $-\infty$;
        CentroidAssigned =0;
        for *j=1 to k* **do**
            dist = $computeDistance(SG_i, KC_j, BC)$;
            if *($Max < dist$)* **then**
                Max = dist;
                CentroidAssigned = j;
            end
        end
        UpdateCentroid($SG_i, KC_{CentroidAssigned}$);
        AssignmentMap.Put(i,CentroidAssigned);
    end
end
//Update assignments iteratively to improve clustering;
numIterations=0;
while $numIterations < MAX$ **do**
    for *i = to q* **do**
        CurrentAssignment = AssignmentMap.Get(i);
        for *j = 1 to k* **do**
            SwapGain = ComputeGain(i, CurrentAssignment, j);
            if *($SwapGain > 0$)* **then**
                Swap(i, CurrentAssignment, j);
            end
        end
    end
    numIterations++;
end
$\mathcal{B} = BinPackingAlgo(\mathcal{KC})$;
return $\mathcal{B}$;

Algorithm 10: ComputeDistance()

Input : Subgraph SG; Centroid C; Bin Capacity BC
Output: Distance between SG and C
if $|SG \cup C| > BC$ **then**
  return -∞
else
  return $|SG \cap C|$
end

in the centroid at all times (i.e., for each vertex in a centroid, we keep track of how many of the assigned subgraphs contain it).

As with K-Means, we make repeated passes over the list of subgraphs in order to improve the clustering. In the subsequent iterations, for each subgraph, we check if it may improve the solution using the function $ComputeGain()$. If the swap gain is positive, i.e. there is a net decrease in the sum of the size of the centroids involved in the swap, we reassign the subgraph to a different centroid, using the multiplicities to remove it from one centroid and assign it to the other centroid ($Swap()$). Finally the k cluster obtained are packed into bins (or partitions).

Having to choose a value of k a priori is one of the key disadvantages of K-Means. We estimate a value of k based on the subgraph sizes and the bin capacity. If at the end of first iteration, we discover that we are left with too many unassigned subgraphs, we increase the value of k and repeat the process till we are able to find a good clustering.

5.3.3 Handling Very Large Subgraphs

Most machines today, even commodity machines, have large amounts of RAM available, and can easily handle very large subgraphs, including 2-hop neighborhoods of high-degree nodes in large-scale networks. However, in the rare case of a subgraph

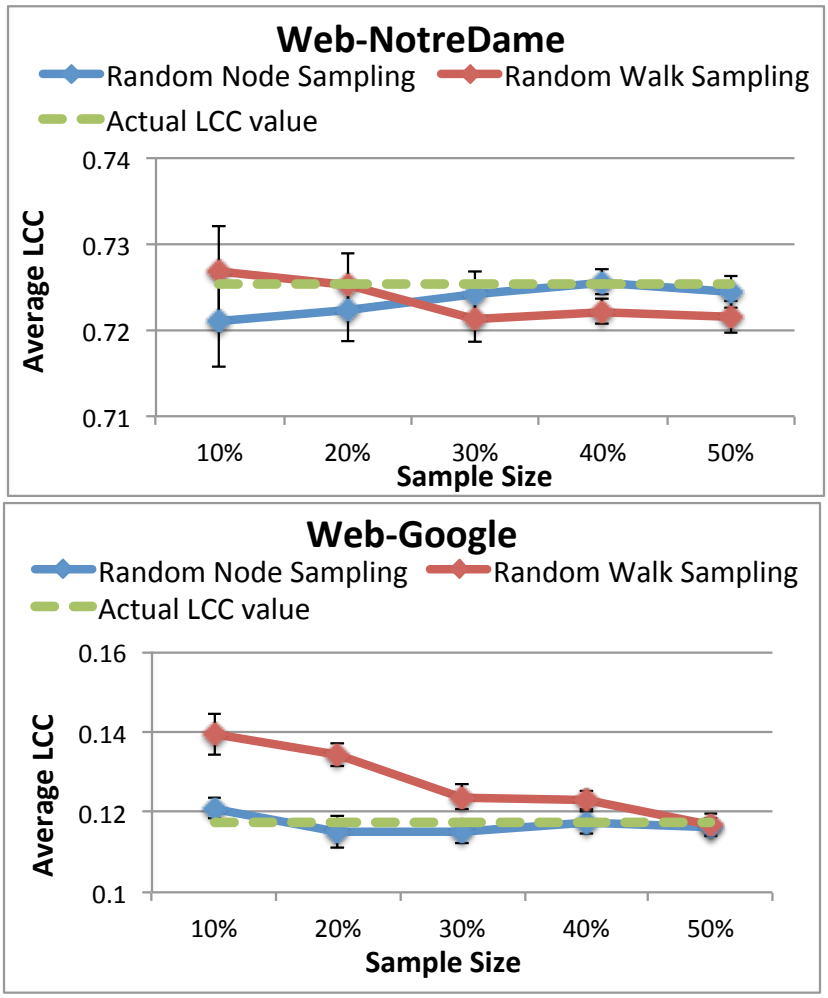


Figure 5.7: **Effect of Graph Sampling**

extraction query where one of the subgraphs extracted is too large to fit into the memory of a single machine, we have two options. The first option is to use disk-resident processing, by storing the subgraph on the disk and loading it into memory as needed. The user program may need to be modified so that it does not thrash in such a scenario. We note here that our flexible programming model makes it difficult to process the subgraph in a distributed fashion (i.e., by partitioning the subgraph across a set of distributed machines); if this scenario is common, we may wish to enforce a vertex-centric programming model within NSCALE, and that is something we plan to consider in future work.

The other option, that we currently support in NSCALE and is arguably better suited for handling large subgraphs, is to use *sampling* to reduce the size of the subgraph. We currently assume that the subgraph skeleton (i.e., the network structure of a subgraph) can be held in the memory of a single machine during GEP; this is needed to support many of the effective random sampling techniques like forest fire or random walks (independent random sampling can be used without making this assumption) [109], [110]. The key idea here is to construct a random sample of a subgraph during GEP, if the size of the subgraph is estimated to be larger than the bin capacity. We provide built-in support for two random sampling techniques: *random node selection*, and *random walk-based sampling*. The former technique chooses an independent random sample of the nodes to be part of the subgraph, whereas the latter technique does random walks starting with the query vertex and including all visited nodes in the sample (till a desired sample size is reached). NSCALE also provides a flexible API for users to implement and provide their own graph sampling/compression technique. The random sampling is performed at the reduce stage in GEP where the subgraph skeleton is first constructed.

Figure 5.7 shows the effect of using our random node and random walk-based sampling algorithms on the accuracy of the local clustering coefficient (LCC) computation. We plot the average LCC computed on samples of different sizes for two different data sets, and compare them to the actual result. Each data point is an average of 10 runs. We also show the standard deviation error bars. For the random node-based sampling techniques, the standard deviation across multiple random runs decreases and the accuracy increases as the sampling ratio increases (as seen in that figure). This is not surprising since the estimated LCC through this technique is an unbiased estimator for the true aver-

age LCC (although it has a very high variance). For the random walk-based sampling, the numbers do not show any consistent trend since the set of sampled nodes does not have any uniformity guarantees and in fact, the set of sampled nodes would be biased towards the high degree nodes (and the effect on the estimated LCC would be arbitrary since the degree of a node is not directly correlated with the LCC for that node).

5.4 Distributed Execution Engine

The NSCALE distributed execution engine runs inside the reduce stage of a MapReduce job (Figure 5.5). The map stage takes as input the original graph and the vertex-to-partition mappings that are computed by the GEP module, and it replicates and shuffles the graph data so that each of the reducers gets the data corresponding to one of the partitions. Each reducer constructs the graph in memory from the data that it receives, and identifies the subgraphs owned by it (the vertex-to-partition mappings contain this information as well). It then uses a worker thread pool to execute the user computation on those subgraphs. The output of the graph computation is written to HDFS.

5.4.1 Execution modes

The execution engine provides several different execution modes. The *vector bitmap mode* associates a bit-vector with each vertex and edge in the partition graph, and enables parallel execution of user computation on different subgraphs. The *batched bitmap mode* is an optimization that uses smaller bitmaps to reduce memory consumption, at the expense of increased execution time. The *single bit bitmap mode* associates a single bit with

each vertex and edge, consuming less memory but allowing for only serial execution of the computation on the subgraphs in a partition.

Vector Bitmap Mode. Here each vertex and edge is associated with a bitmap, whose size is equal to the number of subgraphs in the partition. Each vector bit position is associated with one subgraph and is set to 1 if the vertex or the edge participates in the subgraph computation. A master process on each partition schedules a set of worker threads in parallel, one per subgraph. Each worker thread executes the user computation on its subgraph, using the corresponding bit to control what data the user computation sees. Specifically, our BluePrints API implementation interprets the bitmaps to only return the elements (vertices or edges or attributes) that the callee should see. The use of bitmaps thus obviates the need for state duplication and enables efficient parallel execution of user computation on subgraphs. For consistent and deterministic execution of the user computation, each worker thread can only update the state of the query-vertex contained in its subgraph. We discuss the details of this consistency mechanism in greater detail in Section 5.4.3.

Figure 5.8 shows an example bitmap setting for the subgraphs extracted in Figure 5.3. In Bin 2, subgraphs 2 and 3 share nodes 6 and 7 which have both the bits in the vector bitmap set to 1 indicating that they belong to both the subgraphs. All other nodes in the bins have only one of their bits set, indicating appropriate subgraph membership.

Batching Bitmap Mode. As the system scales to a very large number of subgraphs per reducer, the memory consumed by the bitmaps can grow rapidly. At the same time, the maximum parallelism that can be achieved is constrained by the hardware configuration,

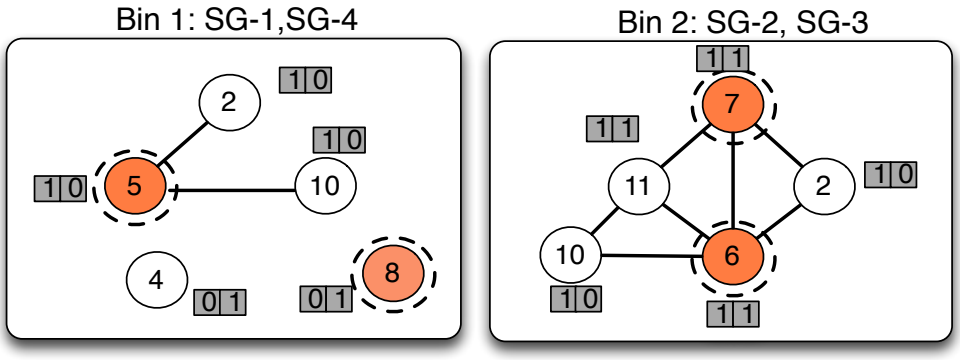


Figure 5.8: **Bitmap based parallel execution**

and it is likely that only a small number of subgraphs can actually be processed in parallel. The batching bitmap mode exploits this by limiting up front the number of subgraphs that may be processed in parallel. Specifically, we batch the subgraphs into batches of a fixed size (called *batch-size*), and process the subgraphs one batch at a time. A bitmap of length *batch-size* is sufficient now to indicate to which subgraphs in the batch a vertex or a node contributes. After a batch is finished, the bitmaps are re-initialized and the next batch commences.

The key question is how to set the batch size. A small batch size may impact the parallelism and may lead to an increased total execution time. A small batch size is also susceptible to the *straggler effect*, where the entire batch completion is held up for one or a few subgraphs (leading to wasted resources and low utilization). A very large batch size, on the other hand, can lead to high memory overheads for negligible reductions in total execution time.

Figures 5.9(a) and 5.9(b) show the results of a set of experiments that we ran to understand the effect of batch size on total execution time and the amount of memory consumed. As we can see, a small batch size indeed leads to underutilization of the

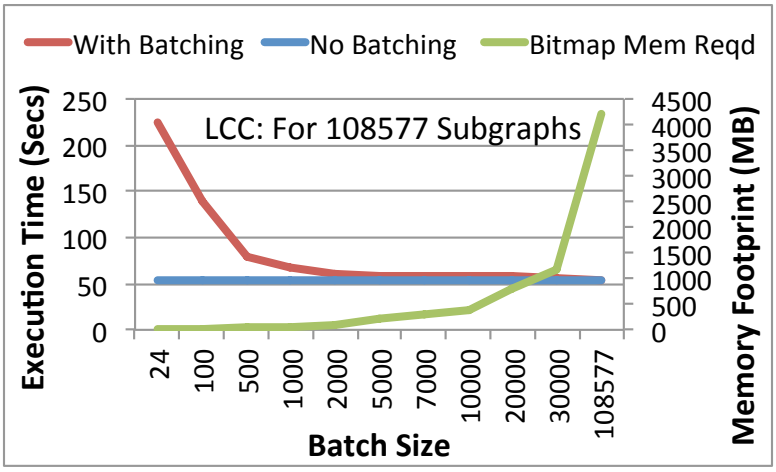
available parallelism and consequently higher execution times. However, we also observe that beyond a certain value, increasing the batch size further did not lead to significant reduction in the execution time. We do a small penalty for batching that can be attributed to the overhead of reinitializing bitmaps across batched execution and to minor straggler effects. However, there is a wide range of parameter values where the execution time penalty is acceptable, and the total memory consumed by the bitmaps is low. Based on our evaluation, we set the batch size to be 3000 for most of our experiments; a lower number should be used if the hardware parallelism is lower (these experiments were done on a 24-core machine), and a higher number is warranted for machines with more cores.

Single-Bit Mode. To further reduce the memory overhead associated with bit vectors, we provide a single bit execution mode wherein each node and edge is associated with a single bit which is set if the node participates in the current subgraph computation. The subgraphs are processed in a serial order, one at a time, with the bits re-initialized after each computation is finished. This mode is supported to cater to distributed computation on low end commodity machines, but it is not expected to scale to large graphs.

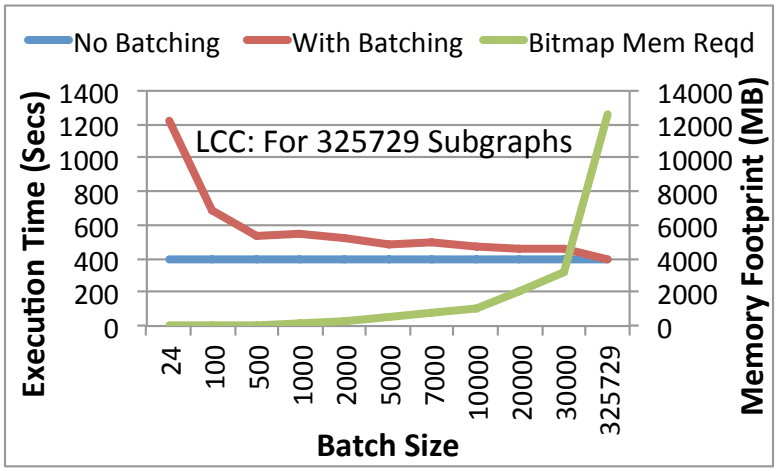
5.4.2 Bitmap Implementation

Given the central role played by bitmaps in our execution engine, we carefully analyzed and compared different bitmap implementations that are available for use in NSCALE.

Java BitSet. Java BitSet Java provides a standard BitSet class that implements a vector of bits that grows as needed. The Java BitSet class internally uses an array of ”longs”



(a)



(b)

Figure 5.9: **Effect of batching on execution time and memory footprints on two different graph datasets.**

and provides generic functionality while maintaining some additional state in terms of an integer and a boolean. The overhead of this extra state is ignorable for large bitmap sizes (5000 and above).

LBitSet. LBitSet To reduce the memory overhead of the Java BitSet class, we implemented the LBitSet class using an array of ”Longs” which actually takes less space than an array of ”longs” if values stored in the array are small. Depending on the bitmap size, an appropriate size of array is chosen. To set a bit, the Long array is considered as a con-

Bitmap size	Java BitSet	L BitSet	C BitSet (Init)	C BitSet (1)	C BitSet (2)	C BitSet (25%)
70	54	39	134	138	142	204
144	63	39	134	138	142	278
3252	484	254	134	138	142	3386
5000	632	321	134	138	142	5134

Table 5.1: **Memory footprints in Bytes for different bitmap constructions and bitmap sizes in bits. For CBitSet, the table shows the initial memory footprint and how it increases when 1 bit is set, 2 bits are set and 25% bits are set (#bits set indicate the #subgraphs the vertex is part of).**

tiguous set of bits and the appropriate bit position is set to 1 using binary bit operations. To unset a bit, the corresponding bit index position is set to 0.

CBitSet. The CBitSet Java class has been implemented using hash buckets. Each bit index in the bitmap hashes (maps) to a unique bucket which contains all the bitmap indexes that are set to 1. To set a bit, the bit index is added to the corresponding hash bucket. To unset a bit, the bit index is removed from the corresponding hash bucket if it is present. This bitmap construction works on the lines of set association, wherein we can hash onto the set and do a linear search within it, thereby avoiding allocation of space of all bits explicitly.

We conducted a micro-benchmark comparing these bitmap implementations to get an estimate of the memory overhead for each bitmap, using a memory mapping utility. Table 5.1 gives an estimate of the memory requirements per node for each of these bitmaps. Memory footprints for CBitSet shown in the table include a column for the initial allotment when the bitmaps are initialized. At run time, when bits are set, this would increase (by about 4 bytes per bit set). The table shows the increase in CBitSet memory as 1, 2, and 25% bits are set. The number of bits set in each bitmap is indicative of the overlap among

them. As we can see, CBitSet would have a lesser memory footprint if the overlap is less. In other cases LBitSet has the least memory footprint. A more detailed performance evaluation of the different bitmap implementations can be found in Section 5.6.3.

5.4.3 Support for Iterative Computation.

NSCALE can naturally handle iterative tasks as well where information must be exchanged across subgraphs between iterations. Below we briefly sketch a description of NSCALE's iterative execution model.

Execution model. NSCALE uses the Bulk Synchronous Protocol (BSP), used by Pregel, Giraph, GraphX, and several other distributed graph processing systems. The analysis task is executed in a number of iterations (also called *supersteps*) with barrier synchronization steps in between the iterations. Since subgraphs of interest typically overlap, the main job of the barrier synchronization step is to ensure that all the updates made by the user program locally to the query vertices are propagated to other subgraphs containing those vertices. During barrier synchronization, after each superstep, the information exchange between subgraphs co-located on the same physical partition is done through shared state updates (saving the overhead of message passing). Information exchange between subgraphs on different physical partitions is done using message passing which is amenable to optimizations such as batching of all updates for a particular partition together, to reduce the overhead.

Consistency model. To provide deterministic execution of iterative computation, the updating of state is closely linked to the query-vertex ownership in NSCALE. Each partition

in NSCALE owns a disjoint set of query-vertices and each worker thread is responsible for one query-vertex and its neighborhood. We only allow updating the state of the query-vertex in each subgraph by the worker thread that owns (or is currently associated with) the query vertex. The state of the query-vertex updated in the current superstep is available for consumption by other subgraphs in the next superstep. This BSP-based consistency model thus does away with the requirement of any explicit locking-based synchronization and its associated overheads making the system easy to parallelize and scalable for large graphs.

We note that, this restriction on the consistency model is equivalent to the restrictions imposed by the other vertex-centric graph processing frameworks, and does not preclude any iterative execution task that we are aware of.

Implementation details. The barrier synchronization required by the BSP execution model can be achieved using any mechanism for reliably maintaining centralized state that can be accessed by different partitions (e.g., one option on YARN is Zookeeper). Further, the message passing model for information exchange between partitions can be built using an in-memory distributed and fault tolerant key-value store like Cassandra [111] or a distributed in-memory key-value cache such as Redis [112], as we do not envision the messages to be very large. The number of components (or partitions) of the distributed key-value store (or cache) can be set equal to the number of partitions in NSCALE with one component co-located with each partition to minimize the network overhead. Each query vertex would mark its updated state in the key-value store that is co-located with the partition to which the query vertex belongs, keyed by the query-vertex ID.

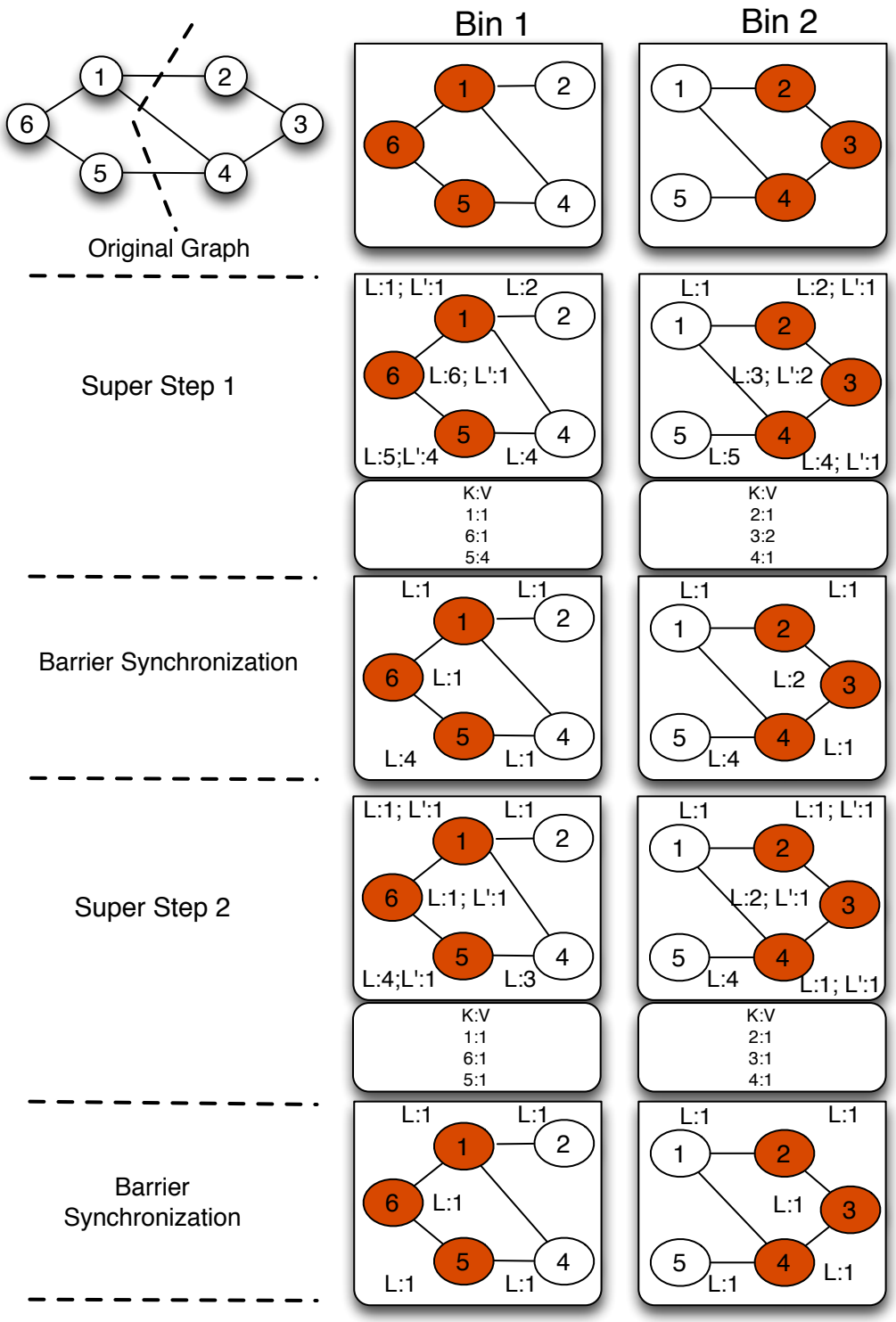


Figure 5.10: **Iterative execution of global connected components algorithm on an example graph on NSCALE.**

In our current implementation, we use Redis for both barrier synchronization using a counter and for message passing. We explain the step-by-process with an example for computing global connected components. Note that, for this application, each vertex in the graph is a query vertex and the set of its 1-hop neighbors constitutes a subgraph of interest.

Example. Figure 5.10 shows an example execution of the global connected components algorithm using multiple supersteps. The figure shows an input graph with vertex IDs as labels of vertices. The GEP phase in NSCALE extracts the subgraphs for each query vertex and instantiates them in two bins (Bin 1 and 2) in an overlapped fashion. Each partition is associated with a disjoint set of query-vertices that it owns. The colored vertices are the query vertices and the other vertices are copies created to enforce the 1-hop neighborhood guarantee. A key-value store shard is also co-located with each partition. Every vertex has an initial label value L (its vertex ID).

In superstep 1, each query vertex accesses the labels of its one-hop neighbors and computes the minimum label and assigns a new value to its own label; the new label is stored in a temporary copy denoted L'. Also each query vertex inserts an entry in the local shard of the distributed K-V store with its ID as the key and its new state (L') as the value. Superstep 1 is followed by barrier synchronization during which the updated values in L' are copied into L for each query vertex, and all non query-vertices in the partition are updated with the values in the distributed key-value store. This is where the message passing takes place between partitions, which is handled by the distributed key-value store under the hood. For improved performance, we use multiple threads to read and write to the Redis key-value cache. In superstep 2, each query vertex repeats

the same procedure and updates its L' values and the key-value store entries. In the subsequent barrier synchronization phase, all the vertices converge to the same label hence terminating the iterations.

5.5 Experimental Evaluation

We performed an extensive experimental evaluation of different design facets of NSCALE and also compared it with three popular distributed graph programming platforms. We briefly discuss some additional implementations details of NSCALE here, and describe the experimental setup.

Implementation Details. NSCALE has been written in Java (version "1.7.0_45") and deployed on a YARN cluster. The framework implements and exports the generic BluePrints API to write graph computations. The GEP module takes the subgraph extraction query, the bin packing heuristic to be used, the bin capacity, and an optional parameter for graph compression/sampling (if required). The YARN platform distributes the user computation and the execution engine library using the distributed cache mechanism to the appropriate machines on the cluster. The execution engine has been parametrized to vary its execution modes, and use different batch sizes and bitmap construction techniques. Although NSCALE has been designed for the cloud, its deployability and design features are not tied to any cloud-specific features; it could be deployed on any cluster of machines or a private cloud that supports YARN or Hadoop as the underlying data-computation framework.

Data Sets. We conducted experiments using several different datasets, majority of which have been taken from the Stanford SNAP dataset repository [113] (see Table 5.2 for details

Dataset	# Nodes	# Edges	Avg Degree	Avg Clust Coeff	# Triangles	Diameter
EU Email Comn Network	265214	840090	3.16	0.0671	267313	14
Notre Dame Web Graph	325729	2,994,268	9.19	0.2346	8910005	46
Google Web Graph	875713	10,210,078	11.66	0.5143	13391903	21
Wikipedia Talk Network	2,394,385	10,042,820	4.2	0.0526	9203519	9
LiveJournal Social Network	4,847,571	137,987,546	28.5	0.2741	285730264	16
Orkut Social Network	3,072,441	234,370,166	76.3	0.1666	627584181	9
ClueWeb Graph	428,136,61	31,448,223,018	3.38	0.2655	4372668765	11

Table 5.2: **Dataset Statistics**

and some statistics).

- **Web graphs:** We have used three different web graph datasets: *Notre Dame Web Graph*, *Google Web Graph*, and *ClueWeb09 Dataset*; in all of these, the nodes represent web pages and directed edges represent hyperlinks between them.

- **Communication/Interaction networks:** We use: (1) *EU Email Communication Network*, generated using email data from a European research institution for a period from October 2003 to May 2005; and (2) The *Wikipedia Talk network*, created from the talk pages of registered users on Wikipedia until Jan 2008.

- **Social networks:** We also use two social network datasets: the *Live Journal social network* and *Orkut social network*.

- **Small-scale synthetic graphs.** For comparing against the optimal algorithm, we generated a set of small-scale synthetic graphs (100-1000 nodes, 500-20000 edges)

using the Barabasi-Albert preferential attachment model.

Graph Applications. We evaluate NSCALE over 6 different applications. Three of them, namely, Local Clustering Coefficient (LCC), Motif Counting: Feed-Forward Loop (MC), and Link Prediction using Personalized Page Rank (PPR), are described in Section **??**. In addition, we used:

- **Triangle Counting (TC):** Here the goal is to count the number of triangles each vertex is part of. These statistics are very useful for complex network analysis [114] and real world applications such as spam detection, link recommendation, etc.

- **Counting Weak Ties (WT):** A weak tie is defined to be a pattern where the center node is connected to two nodes that are not connected to each other. The goal with this task is to find the number of weak ties that each vertex is part of. Number of weak ties is considered an important metric in social science [115].

In addition to the above graph applications that involve single-pass analytics, we also evaluated NSCALE using a global iterative graph application, computing the connected components, as described in Section 5.4.3.

Comparison platforms. We compare NSCALE with three widely used graph programming frameworks.

- **Apache Giraph [85].** The open source version of Pregel, written in Java, is a vertex-centric graph programming framework and widely used in many production systems (e.g., at Facebook). We deploy Apache Giraph (Version 1.0.0) on Apache YARN with Zookeeper for synchronization for the BSP model of computation. Deploying Apache

Giraph on YARN with HDFS as the underlying storage layer enables us to provide a fair comparison using the same datasets and graph applications.

- **GraphLab [116].** GraphLab, a distributed graph-parallel API written in C++, is an open source vertex-centric programming model that supports both synchronous and asynchronous execution. GraphLab uses the GAS model of execution wherein each vertex program is decomposed into *gather, apply*, and *scatter* phases; the framework uses MPI for message passing across machines. We deployed GraphLab v2.2 which supports OpenMPI 1.3.2 and MPICH2 1.5, on our cluster.
- **GraphX [59].** GraphX is a graph programming library that sits on top of Apache Spark. We used the GraphX library version 2.10 over Spark version 1.3.0 which was deployed on Apache YARN with HDFS as the underlying storage layer.

Evaluation metrics. We use the following evaluation metrics to evaluate the performance of NSCALE.

- **Computational Effort** ($\mathcal{CE}$). $\mathcal{CE}$ captures the total cost of doing analytics on a cluster of nodes deployed in the cloud. Let $T = \{T_1, T_2, ..., T_N\}$ be the set of tasks (or processes) deployed by the framework on the cluster during execution of the analytics task.

  Also, let t_i be the time taken by the task T_i to be executed on node i. We define $\mathcal{CE} = \sum_{i=1}^{N} t_i$. The metric captures the cost of doing data analytics in terms of *node-secs* which is appropriate for the cloud environment.

- **Execution Time.** This is the measure of the wall clock time or elapsed time for executing an end-to-end graph computation on a cluster of machines. It includes

the time taken by the GEP phase for extracting the subgraphs as well as the time taken by the distributed execution engine to execute the user computation on all subgraphs of interest.

- **Cluster Memory.** Here we measure the maximum total physical memory used across all nodes in the cluster.

Experimental Setup. We use two 16 node clusters wherein each data node has 2 4-core Intel Xeon E5520 processors, 24GB RAM and 3 2 TB disks. The first cluster runs Apache YARN (MRv2 on Cloudera's CDH version 5.1.2) and Apache Zookeeper for coordination. Each process on this cluster runs in a container with a max memory capacity restricted to 15GB with a maximum of 6 processes per physical machine. We run NSCALE, Giraph and GraphX experiments on this cluster. The second cluster supports MPI for message passing and uses a TORQUE (Terascale Open-Source Resource and QUEue) Manager. We run GraphLab in this cluster and restrict the max memory per process on each machine to 15GB for a fair comparison.

For all our baseline comparisons and scalability experiments, we have used the shingle-based bin packing heuristic as the GEP algorithm for packing subgraphs into bins. We have chosen shingle-based bin packing as it finds good quality solutions efficiently, while consuming fewer resources as compared to the other heuristics. Also, for smaller graphs such as NotreDame web graph, Google web graph, etc., where the filtered structure can fit onto a single machine, we used the centralized GEP solution (Ref Case 1, Section 5.3.1). On the other hand, for larger graphs such as the Clue Web graph, we use the distributed GEP solution (Ref Case 2 Section 5.3.1).

5.6 Experimental Results

5.6.1 Baseline Comparisons

We begin with comparing NSCALE with Apache Giraph and GraphLab for different datasets for the five different applications. For four of the applications (LCC, MC, TC, WT), the subgraphs of interest are specified as 1-hop neighborhoods of a set of query vertices which could be chosen randomly or specified using query-vertex predicates. On the other hand, Personalized Page Rank (PPR) is computed on the 2-hop neighborhood of a set of query vertices. For a fair comparison with all the other baselines, we choose each vertex as a query-vertex for NSCALE and run the the first four applications (LCC, MC, TC, WT) on their 1-hop neighborhoods in a single pass. For the Personalized page rank application we choose different number of source (or query) vertices for different datasets. The Personalized page rank is computed with respect to these source vertices on their 2-hop neighborhoods in all frameworks.

Tables 5.3 and 5.4 show the results for the baseline comparisons. Since all of these applications require access to neighborhoods, Apache Giraph runs them using multiple iterations. In the first superstep it gathers neighbor information using message passing and in the second superstep, it does the required graph computation (for PPR, Giraph needs two supersteps to gather the 2-hop neighborhoods).

As we can see, for most of the graph analytics tasks, Giraph does not scale to larger graphs. It runs out of memory (OOM) a short while into the map phase, and does not complete (DNC) the computation.

	Local Clustering Coefficient							
Dataset	NSCALE		Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)
EU Email	377	9.00	1150	26.17	365	20.1	225	4.95
NotreDame	620	19.07	1564	30.14	550	21.4	340	9.75
GoogleWeb	658	25.82	2024	35.35	600	33.5	1485	21.92
WikiTalk	726	24.16	DNC	OOM	1125	37.22	1860	32
LiveJournal	1800	50	DNC	OOM	5500	128.62	4515	84
Orkut	2000	62	DNC	OOM	DNC	OOM	20175	125

	Motif Counting: Feed-Forward Loop							
Dataset	NSCALE		Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)
EU Email	279	8.76	1371	24.43	285	20.8	4125	7.2
NotreDame	524	18.02	1923	28.98	575	21.6	10875	15.6
GoogleWeb	812	23.64	2164	37.27	625	31.9	DNC	-
WikiTalk	991	29.34	DNC	OOM	1150	36.81	DNC	-
LiveJournal	1886	51	DNC	OOM	4750	130.74	DNC	-
Orkut	2024	63	DNC	OOM	DNC	OOM	DNC	-

	Per-Vertex Triangle Counting							
Dataset	NSCALE		Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)
EU Email	264	15.36	1012	26.10	250	21.1	240	4.5
NotreDame	477	17.62	1518	30.16	425	22.7	270	9
GoogleWeb	663	25.86	1978	35.39	550	31.3	1230	21
WikiTalk	715	21.29	DNC	OOM	975	32.22	1590	30.2
LiveJournal	1792	49.34	DNC	OOM	4750	129.61	4335	74
Orkut	1986	61.32	DNC	OOM	DNC	OOM	13875	115

Table 5.3: **Comparing NSCALE with Giraph, GraphLab and GraphX**

	Identifying Weak Ties							
Dataset	NSCALE		Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)
EU Email	278	7.34	1472	25.49	281	20.4	4215	7.3
NotreDame	390	13.26	2024	29.99	400	20.6	11795	16.6
GoogleWeb	555	21.60	2254	39.26	525	30.7	DNC	-
WikiTalk	592	18.18	DNC	OOM	925	31.71	DNC	-
LiveJournal	1762	48.32	DNC	OOM	4625	126.71	DNC	-
Orkut	1972	60.45	DNC	OOM	DNC	OOM	DNC	-

	Personalized Page Rank on 2-hop Neighborhood								
Dataset		NSCALE		Giraph		GraphLab		GraphX	
	#Source Vertices	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem (GB)
EU Email	**3200**	52	3.35	782	17.10	710	28.87	9975	85.5
NotreDame	**3500**	119	9.56	1058	31.76	870	70.54	50595	95
GoogleWeb	**4150**	464	21.52	10482	64.16	1080	108.28	DNC	-
WikiTalk	**12000**	3343	79.43	DNC	OOM	DNC	OOM	DNC	-
LiveJournal	**20000**	4286	84.94	DNC	OOM	DNC	OOM	DNC	-
Orkut	**20000**	4691	93.07	DNC	OOM	DNC	OOM	DNC	-

Table 5.4: **Comparing NSCALE with Giraph, GraphLab and GraphX**

Hence these baseline comparisons have been shown on relatively smaller graphs. The cluster logs confirmed that the poor scalability of Giraph for such applications is due to the high message passing overhead between the vertices, characteristic of vertex-centric approaches like Giraph, and high memory requirements due to duplication of state at each vertex.

Compared to Giraph, GraphLab performs a little better. For smaller graphs such as NotreDame and Google Web, GraphLab's performance is comparable to NSCALE and for some applications like Local Clustering Coefficient, it is a little better than NSCALE in terms of $\mathcal{CE}$. However, in all cases, GraphLab consumes much more cluster memory depending on the graph partitioning mechanism and the replication factor it uses, the latter

of which varies with the number of machines on which the job is executed. Like Giraph, GraphLab too does not scale to larger graphs for neighborhood-centric applications.

GraphX does well for 1-hop graph applications such as LCC and TC on smaller graphs both in terms of memory and $\mathcal{CE}$ (node-secs). However as the graph size increases, $\mathcal{CE}$ grows rapidly and surpasses that of NSCALE, quite significantly. For applications such as MC and WT, GraphX performs poorly as these applications require explicit edge information between the 1-hop neighbors of the query-vertex which necessitates joins and triplet aggregations across the vertex and edge RDDs, leading to poor scalability for larger graphs for such applications. For similar reasons, the performance of GraphX further deteriorates for 2-hop neighborhood applications such as PPR and it does not complete for any of the larger graph datasets (Web-Google and beyond).

Tables 5.5 and 5.6 show the performance gain of NSCALE, over Giraph, GraphLab and GraphX both in terms of $\mathcal{CE}$ and cluster memory consumption. Even for the smaller graphs, depending on the type of application and the size of neighborhood, NSCALE performs 3X to 22X better in terms of $\mathcal{CE}$, and consumes a lot less (up to 5X less) total cluster memory as compared to Giraph.

GraphLab follows a similar trend. As can be seen, for all the five applications, as the graph size increases, both $\mathcal{CE}$ and required memory increase sharply, and GraphLab fails to complete, running out of memory, for real world graphs such as WikiTalk, Orkut and Live Journal. Even for relatively smaller graphs, the performance difference is significant, especially for 2-hop applications such as Personalized Page Rank where GraphLab is up to 13X slower and consumes up to 8X more memory.

	Local Clustering Coefficient					
Dataset	Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)
EU Email	3.05X	2.9X	0.96X	2.23X	0.66X	0.55X
NotreDame	2.52X	1.58X	0.88X	1.12X	0.54	0.51
GoogleWeb	3.07X	1.36X	0.91X	1.29X	2.25X	0.84X
WikiTalk	-	-	1.54X	1.54X	2.56X	1.32X
LiveJournal	-	-	3.05X	2.57X	2.50X	1.68X
Orkut	-	-	-	-	10.08X	2.01X

	Motif Counting: Feed-Forward Loop					
Dataset	Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)
EU Email	4.91X	2.78X	1.02X	2.37X	14.78	0.82X
NotreDame	3.66X	1.60X	1.09X	1.19X	20.75X	0.86X
GoogleWeb	2.66X	1.57X	0.76X	1.34X	-	-
WikiTalk	-	-	1.16X	1.25X	-	-
LiveJournal	-	-	2.51X	2.56X	-	-
Orkut	-	-	-	-	-	-

	Per-Vertex Triangle Counting					
Dataset	Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)
EU Email	3.83X	1.69X	0.94X	1.37X	0.90X	0.29X
NotreDame	3.18X	1.71X	0.89X	1.28X	0.56X	0.51X
GoogleWeb	2.98X	1.36X	0.82X	1.21X	1.85X	0.81X
WikiTalk	-	-	1.36X	1.51X	2.22X	1.41X
LiveJournal	-	-	2.65X	2.62X	2.41X	1.49X
Orkut	-	-	-	-	6.98X	1.87X

Table 5.5: **Performance (X) improvement of NSCALE over Giraph, GraphLab and GraphX; a "-" indicates that the other system ran out of memory or did not complete.**

	Identifying Weak Ties					
Dataset	Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)
EU Email	5.29X	3.47X	1.01X	2.77X	15.16X	0.99X
NotreDame	5.18X	2.26X	1.02X	1.55X	30.24X	1.25X
GoogleWeb	4.06X	1.81X	0.94X	1.42X	-	-
WikiTalk	-	-	1.56X	1.74X	-	-
LiveJournal	-	-	2.62X	2.62X	-	-
Orkut	-	-	-	-	-	-

	Personalized Page Rank					
Dataset	Giraph		GraphLab		GraphX	
	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)	$\mathcal{CE}$ (Node-Secs)	Cluster Mem(GB)
EU Email	15.03X	5.10X	13.65X	8.61X	191.82X	25.52X
NotreDame	8.89X	3.32X	7.31X	7.37X	425.16X	9.93X
GoogleWeb	22.59X	2.98X	2.32X	5.03X	-	-
WikiTalk	-	-	-	-	-	-
LiveJournal	-	-	-	-	-	-
Orkut	-	-	-	-	-	-

Table 5.6: **Performance (X) improvement of NSCALE over Giraph, GraphLab and GraphX; a "-" indicates that the other system ran out of memory or did not complete.**

GraphX performs better for smaller graphs for applications such as LC and TC. However, for relatively larger graphs, NSCALE is up to 10X better in terms of $\mathcal{CE}$ and consumes up to 2X less memory. For MC and WT applications, NSCALE is up to 30X better in terms of $\mathcal{CE}$ and consumes up to 1.25X less memory for smaller graphs. For larger graphs GraphX fails to complete. The most significant difference is seen for PPR where NSCALE performs up to 425X better in terms of $\mathcal{CE}$ and consumes up to 25X lesser memory for smaller graphs. Again, for larger graphs GraphX fails to complete.

The improved performance of NSCALE can be attributed to the NSCALE computation and execution models which (1) allow a user computation to access the entire subgraph state and hence do not require multiple iterations avoiding the message passing

overhead, and (2) avoid duplication of state at each vertex reducing memory requirements drastically. Further, the extraction and loading of required subgraphs by the GEP module helps NSCALE to scale to larger graphs using minimal resources.

5.6.2 GEP Evaluation

Comparing subgraph bin packing (SBP) algorithms. We first evaluated the the performance and quality of the bin packing-based algorithms: First Fit, First Fit Decreasing and Shingle Based bin packing on the LiveJournal data set.

Figure 5.11(a) shows the number of bins required to partition the subgraphs as we vary the number of subgraphs specified by the query (using predicates on the query vertices). The number of bins increases as the number of subgraphs increases for all the three heuristics. We see that the First Fit algorithm requires the maximum number of bins as expected whereas the shingle-based packing algorithm performs the best in terms of packing the subgraphs into a minimum number of bins. This is due to the fact that the shingle-based bin packing algorithm orders the subgraphs based on neighborhood similarity thereby taking maximum advantage of the overlap amongst them.

To ascertain the cost of data analytics we study the effect of bin packing on the computation effort $\mathcal{CE}$. Figure 5.11(b) shows that the $\mathcal{CE}$ for the First Fit algorithm is the maximum making it the most expensive, while the $\mathcal{CE}$ for shingle-based packing algorithm is the minimum making it the most cost effective bin packing solution. Figures 5.11(c), 5.11(d) show the execution (elapsed) time and the total cluster memory usage for binning and execution with respect to these three heuristics and different number of subgraphs. The First Fit has the best execution time and the shingle-based bin packing algorithm has

an execution time which closely follows that of the First Fit algorithm. On the other hand, the First Fit Decreasing algorithm takes the largest execution time. This can be attributed to the fact the First Fit is expected to produce the most uniform distribution of subgraphs across the bins and the First Fit Decreasing is likely to produce a skewed distribution (packing a large number of smaller subgraphs in later bins) leading to larger execution times due to the straggler effect.

We further study the distribution of the number of subgraphs (or query-vertices) packed per bin and the distribution of running times of each instance of a execution engine on a bin (partition). Figures 5.11(e),5.11(f) show the box plots with whiskers for both the distributions. As expected the First Fit algorithm has the most uniform distribution across the bins in both cases. The shingle-based packing algorithm also performs well and provides a distribution almost as good as the First Fit algorithm, while First Fit Decreasing has the most skewed distribution in both cases, which also explains the highest end-to-end execution timings for the heuristic. We thus see that the shingle-based packing algorithm performs the best in terms of minimizing the # bins and $\mathcal{CE}$, having low execution times and almost uniform bin distributions thus minimizing the straggler effect.

To summarize, our results showed that our shingle-based packing algorithm performs much better in terms of minimizing the # bins and $\mathcal{CE}$. It also has low execution times and almost uniform bin distributions thus minimizing the straggler effect.

We next compare the shingle-based bin packing heuristic with the two clustering-based algorithms, and the METIS-based algorithm. Figures 5.12(a), 5.12(b) and 5.12(c) show the performance of the four subgraph packing approaches for three different real-world datasets.

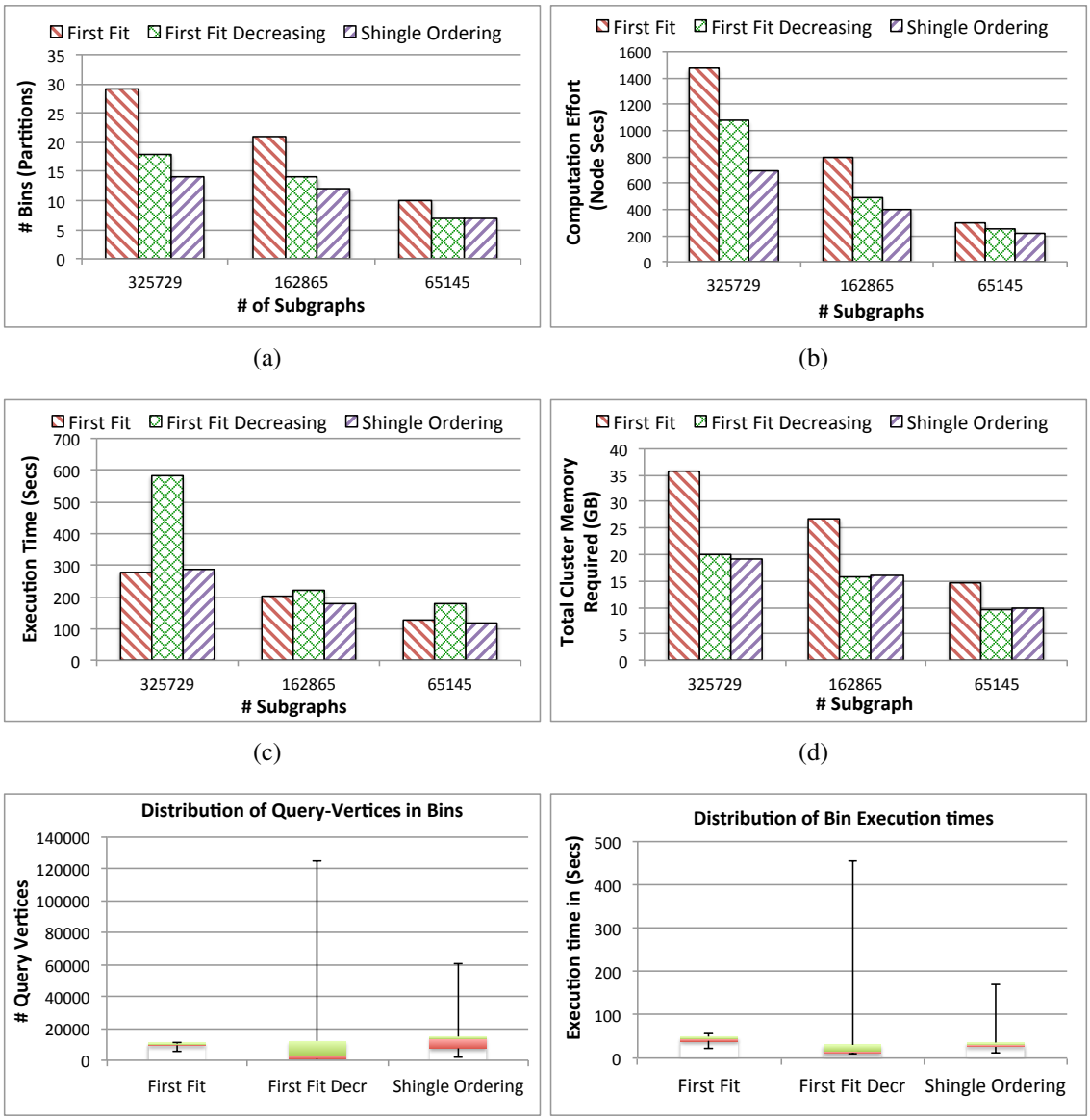


Figure 5.11: **For the different shingle based subgraph packing heuristics, we compare: (a) #bins required; (b) total computational effort required; (c) total elapsed time (wall clock time) for running the LCC computation on the subgraphs; (d) total cluster memory required for GEP and execution of the LCC computation; (e)-(f) distribution of # subgraphs and of execution engine running times over the bins**

We see that K-Means provides generally found solutions with minimum number of bins, but takes much longer and consumes significantly more memory. The shingle-based solution finds almost as good solutions, but is much more efficient. METIS-based

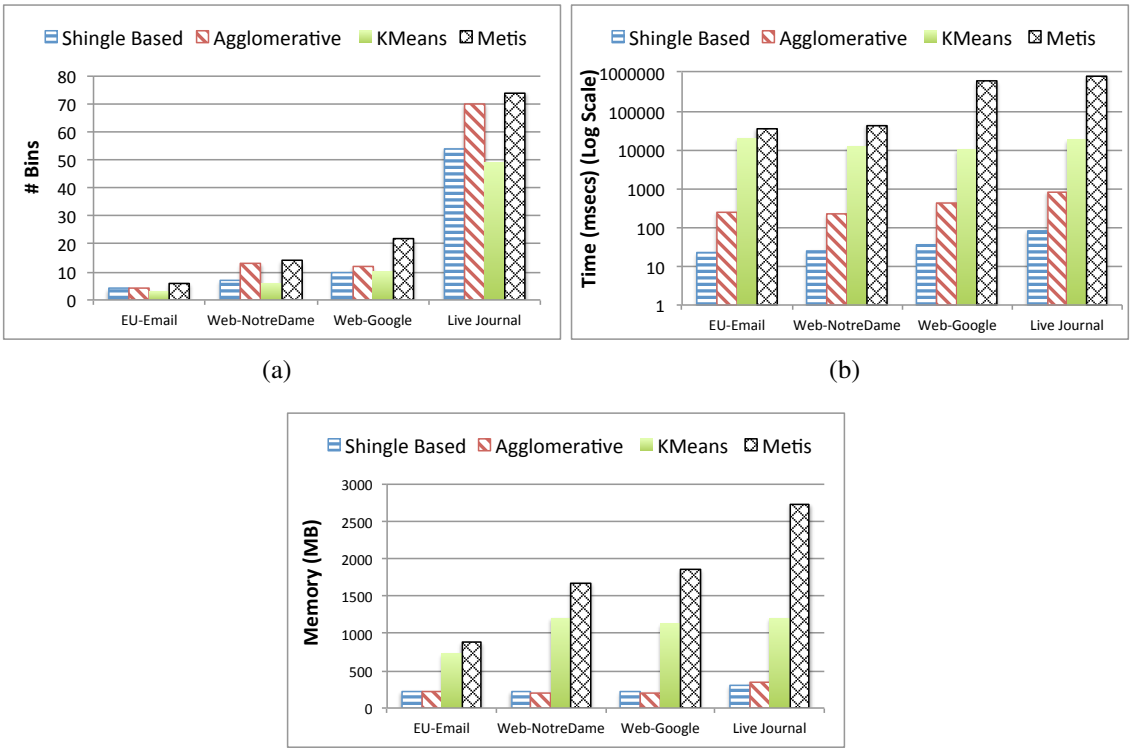


Figure 5.12: **Comparison of shingle based subgraph packing heuristics with the other bin packing heuristics; we compare: (a) #bins required; (b) total time taken for bin packing; (c) memory required.**

partitioning does poorly both in terms of binning quality and the efficiency (notice the log scale in Figure 5.12(b)), and we did not consider it for the rest of experimental evaluation.

To better evaluate the performance of the heuristics, we also compared them with an optimal algorithm (OPT), that constructs an Integer Program for the problem instance, and uses the Gurobi Optimizer to find an optimal solution. Unfortunately, even after many hours on a powerful server per problem instance, OPT was unable to find a solution for most of our small-scale synthetically generated problem instances; for 14 of 64 synthetic datasets, it found either an optimal solution or reasonable bounds, and we have plotted those in Figure 5.13(a) (the x-axis is sorted by the value of the best solution found by OPT). We note that the only instances where OPT found the optimal solution (i.e., where

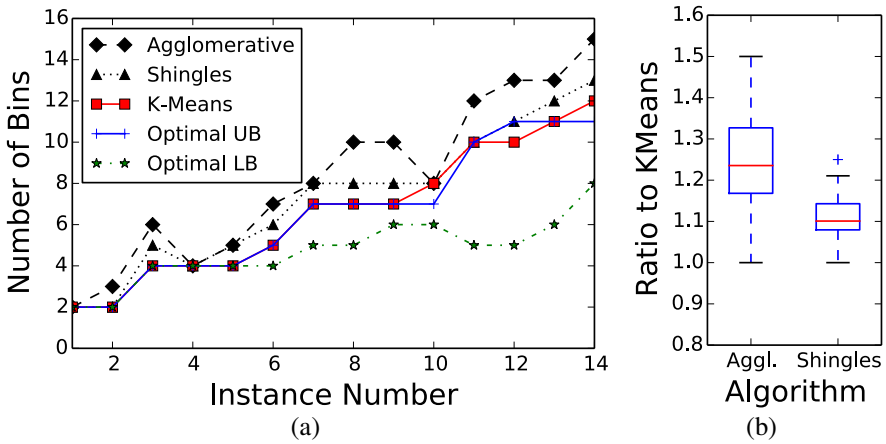


Figure 5.13: **Comparing subgraph packing heuristics to (a) the optimal solution and (b) each other, for synthetic graphs**

upper bound = lower bound) were solutions with 2 or 4 bins. As we can see, for almost all of these problem instances, our K-Means heuristics was able to match the OPT solution.

Overall, the reason K-Means performs so well can be attributed to the fact that it explores the solution space more extensively and in general, does more pair-wise comparisons between the sets (corresponding to the subgraphs). The behavior was consistent across a wide range of experiments that we did. The shingle-based heuristic, on other other hand, restricts the comparisons to subgraphs that are close in the shingle order, and thus may miss out on pairs of sets that have high overlap. At the same time, we want to note that KMeans takes much longer to run and consumes significantly more memory, whereas the shingle-based heuristic is much faster and finds solutions with comparable quality. Figure 5.13(b) compares the K-Means heuristics against the other two heuristics for all 64 datasets. The results are consistent with the results we saw on the real-world datasets – K-Means is consistently better than both of those heuristics, but the shingle-based heuristic comes quite close to its performance.

Distributed GEP evaluation. Figures 5.14(a), 5.14(b), and 5.14(c) compare the distributed implementation of the GEP module with the centralized version (we use LiveJournal dataset for this purpose, which is small enough for a centralized solution, and we use 6 machines in the distributed case).

We see that, for a small number of extracted subgraphs, the time taken by the centralized solution is comparable to the time taken by the distributed solution. However as we scale to a large number of subgraphs, the distributed solution scales much better, and more importantly, the maximum memory required on any single machine is much lower, thus removing a key bottleneck of the centralized solution. The binning quality of the centralized solution is somewhat better, which is to be expected, and hence it would still be preferable to run the GEP phase in a centralized fashion. However the gap is not significant, and for large graphs where running GEP in a centralized fashion is not feasible, distributed GEP generates reasonable solutions.

Figures 5.14(d), 5.14(e) and 5.14(f) show the effect of increasing the number of machines used for distributed GEP on the time taken, memory required per machine and the quality of binning solution provided in terms of number of bins required, for three data sets. We see that our distributed GEP mechanism exhibits good scaling behavior without compromising much on the quality of binning. It can thus handle very large graphs quite effectively. Note that the number of query vertices was set to 3M, so the relative performance for the different graph does not correlate with the original graph sizes (in particular, ClueWeb has low average degree, hence requires fewer bins for the same number of neighborhoods).

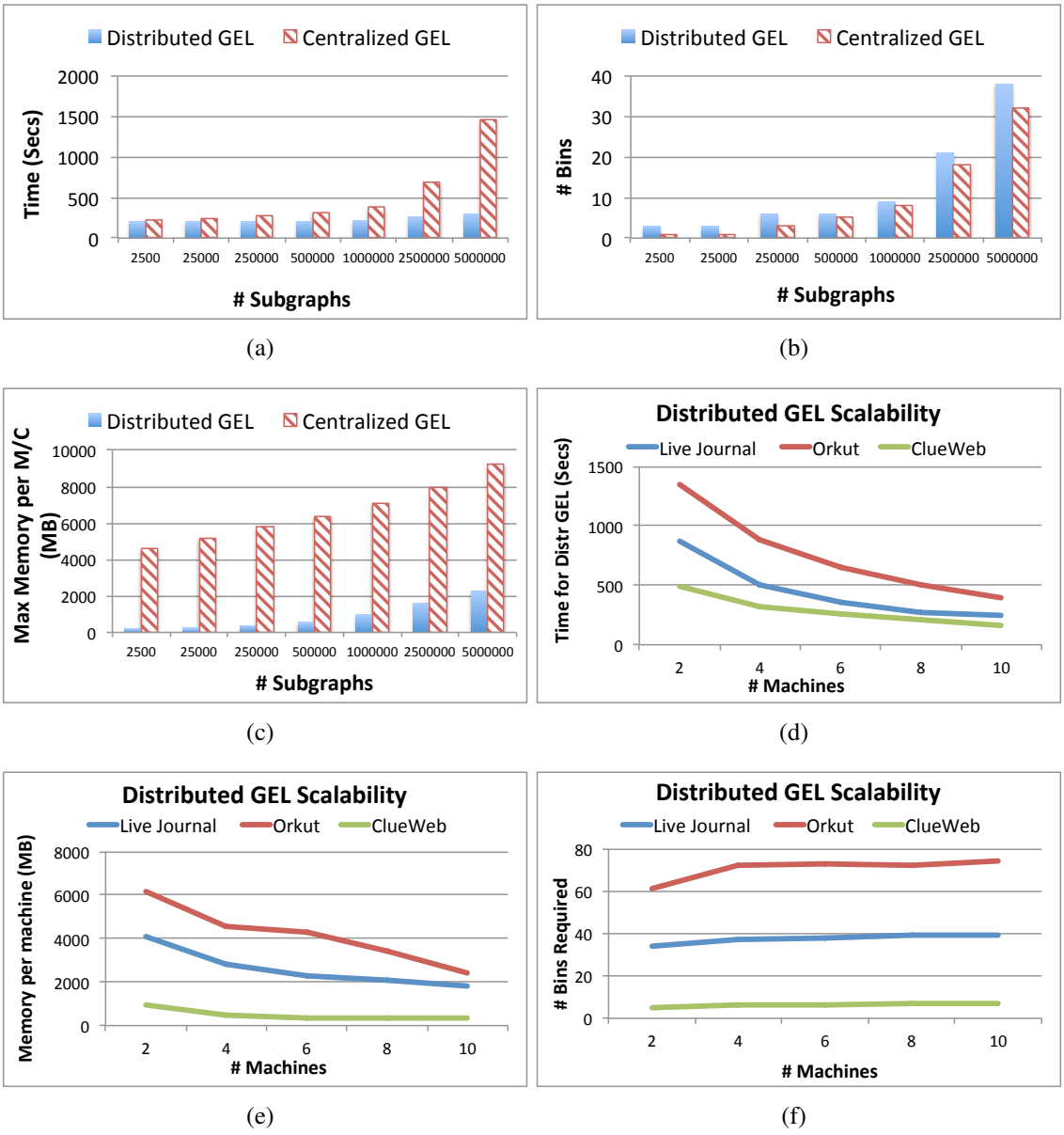


Figure 5.14: **GEP architecture: (a)-(c) Comparison of centralized and distributed GEP architectures; (d)-(f) Distributed GEP architecture: Impact on graph extraction and packing time, max memory required per bin, and #bins required for packing with increase in number of machines.**

5.6.3 Execution Engine Evaluation

Effect of choosing different execution modes. In Figure 5.15(a), we plot the total running times for the single-bit serial (SEM) and vector bitmap parallel (PEM) execution modes, for the LiveJournal graph, for 25000 extracted subgraphs. We see that for 70 partitions, the performance of the two modes is comparable since each partition does a small amount of work. However as the number of partitions decreases, PEM performs much better compared to SEM which times out as the number of partitions becomes very small. On the other hand, SEM uses a single bit bitmap per vertex or edge and hence requires significantly less memory, and may be useful when we have a large number of low memory machines available for graph computation.

Bitmap constructions. Figures 5.15(b), 5.15(c) compare the different bitmap implementations for different numbers of partitions (the setup is the same as above, and hence decreasing number of partitions implies increasing number of subgraphs per partition). Java BitSet and LBitSet perform better than CBitSet in terms of execution time, while LBitSet consumes the least amount of memory as the number of subgraphs in each partition increases. As mentioned in Section 5.4.2, CBitSet is useful in cases where the overlap between subgraphs is minimum, requiring a small number of bits to be set. We use LBitSet for most of our experiments.

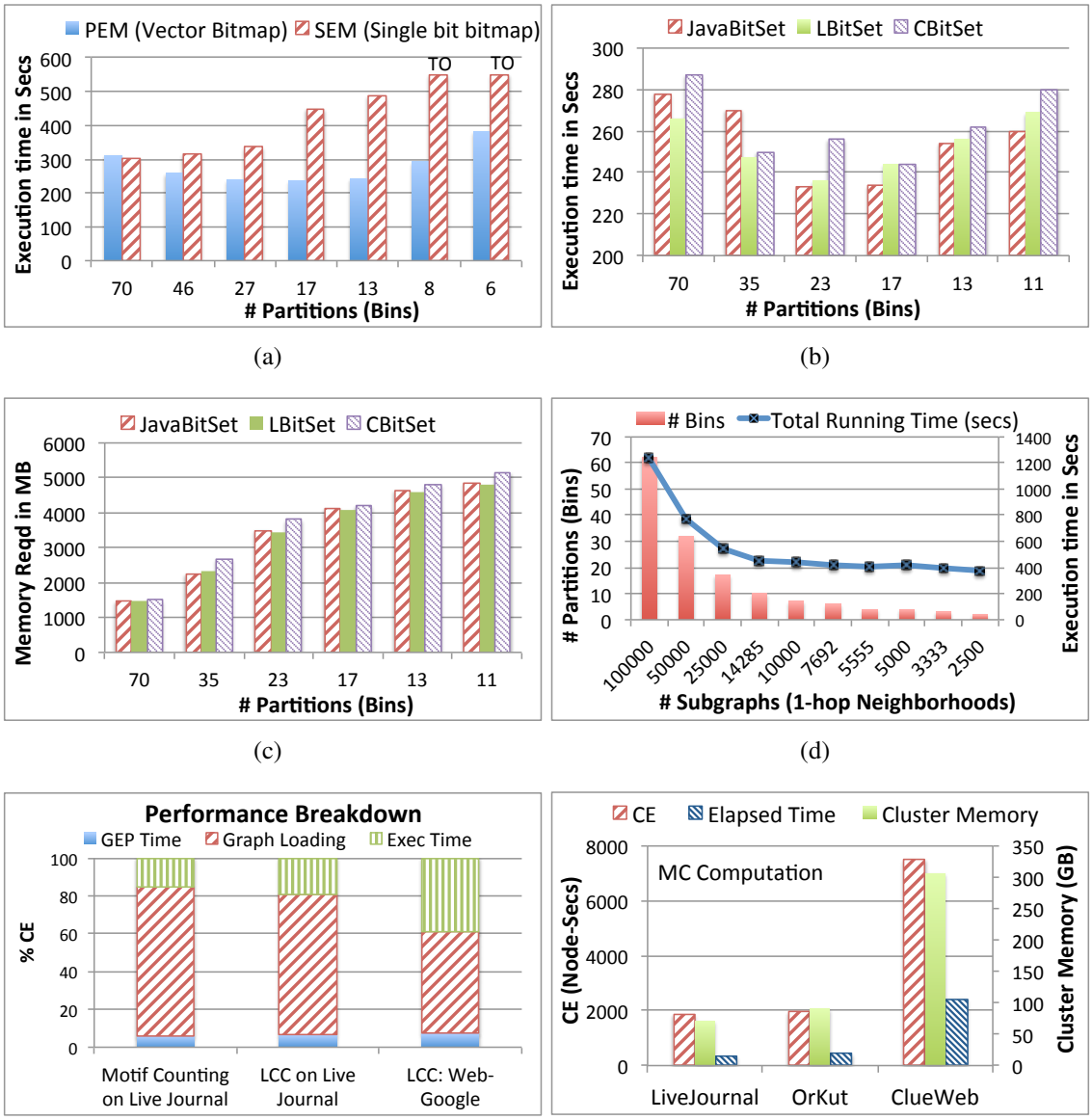


Figure 5.15: **(a) Effect of different execution modes on the running time; (b)-(c) Effect of different bitmap implementations on the memory footprints and the running times of the execution engine; (d) End-to-End running time and #partitions required for different numbers of subgraphs; (e) Performance breakdown of different stages of NSCALE for graphs of different sizes and different applications; (f) Scalability: NSCALE performance over large graphs.**

5.6.4 System Evaluation

End-to-End testing. We evaluate the overall performance of the system for a fixed bin capacity (8GB) for the LiveJournal graph. We vary the number of subgraphs to be extracted from the underlying graph and study the effect on the number of bins required to pack them into memory using the shingle-based bin packing heuristic. We measure the total end-to-end running time of the LCC computation on each of these subgraphs in PEM mode using LBitSet bitmap construction. Figure 5.15(d) shows that as the number of subgraphs increases, the system distributes the computation on a larger number of bins and scales well with respect to the increase in the total running time with increase in number of subgraphs which includes the time required by the GEP phase and the actual graph computation by each instance of the execution engine on each partition.

Performance breakdown. Figure 5.15(e) shows the breakdown in terms of the $\%\mathcal{CE}$ required for the different stages of NSCALE. The figure shows the performance for two different applications: LCC and Motif Counting and two different graphs: LiveJournal and Web-Google. For the smaller graphs like Web-Google, the % time taken for execution is comparable to the graph loading time. For larger graphs like LiveJournal, the % graph loading time dominates all other times as it includes the time taken to read the disk resident graph and associated data, filter and shuffle based on the partitioning obtained from GEP. In all cases, **GEP constitutes a small fraction of the total time and is the crucial component that enables the efficient execution of the graph computation on the materialized subgraphs in distributed memory using minimal resources.** As can be seen in the baseline comparisons, without the GEP phase, other vertex-centric ap-

proaches have a very high $\mathcal{CE}$ as compared to NSCALE for the same underlying datasets and graph computations.

NSCALE performance for larger graphs. To ascertain the scalability of NSCALE we conducted experiments with larger datasets for the Motif Counting application. Figure 5.15(f) shows the results for the scalability experiments on the Social LiveJournal graph, the Orkut social network graph, and the largest of our datasets, the ClueWeb graph (428M nodes, 1.5B edges). The results show the $\mathcal{CE}$ in node-secs and total cluster memory required in GB. The results indicate that NSCALE scales well for ego-centric graph computation applications over larger graphs unlike other vertex-centric approaches such as Apache Giraph and GraphLab.

5.6.5 Evaluation of Support for Iterative Applications.

We evaluated the support for iterative applications using the global connected components application.

Performance breakdown. We studied the performance breakdown for the connected components application across different iterations over two different datasets (LiveJournal and Orkut). Figures 5.16(a), 5.16(b), 5.16(c), and 5.16(d) show the performance breakdown in terms of compute time, synchronization time (time spent waiting at the barriers), and message passing time (time spent in updating the key-value store and fetching updated values from the key-value store). We studied the performance breakdown for two different scenario, where the graph was partitioned across 5 or 10 machines (we adjusted the bin capacity parameter to find the setting which forced NSCALE to use the appropri-

ate number of machines). As expected, with 5 partitions, the synchronization overhead is more as compared to the message passing overhead since the number of ghost vertices that require message passing is smaller. In comparison, with 10 partitions, the message passing overhead is more as the number of ghost vertices is relatively higher. The synchronization overhead is less as each partition does less work and inter-partition skew is smaller.

Performance comparison. Figures 5.16(e) and 5.16(f) compare the performance of NSCALE against GraphX and GraphLab for the connected components application in terms of the running time (Wall Clock time) and $\mathcal{CE}$ (node-secs) on 10 machines. As we can see, our relatively unoptimized implementation compares favorably to both, and in fact, outperforms GraphX in some of the cases. Overall, GraphLab performs better in terms of both runtime and $\mathcal{CE}$ for both graph datasets. This superior performance of GraphLab for iterative computations can be attributed to its highly optimized MPI-based message passing layer, as well as its implementation in C++.

5.6.6 Discussion.

In summary, our comprehensive experimental evaluation illustrates that NSCALE has comparable performance to the other graph processing frameworks for iterative tasks like connected components, while vastly outperforming them for more complex analysis tasks. Although NSCALE is able to scale better than the other systems we compared against for most of the tasks and it uses fewer resources in general, there is certainly a limit to the graph sizes that our current implementation can handle given limited resources,

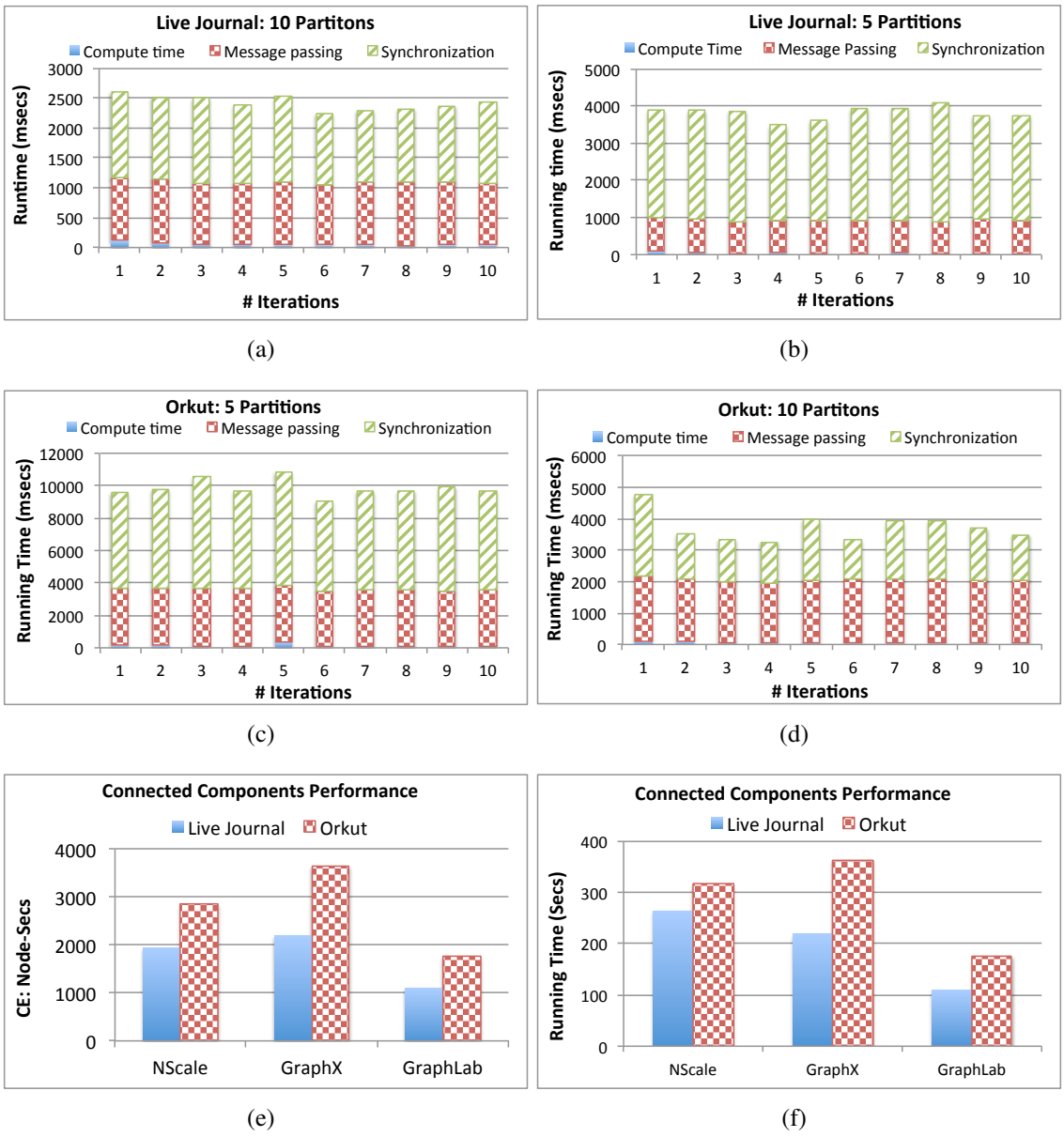


Figure 5.16: **Connected components: (a-d) Performance break down for different iterations; (e-f) Performance comparison with GraphX and GraphLab in terms of running time and CE (node-secs).**

and those can be seen or extrapolated from our reported numbers (e.g., NSCALE wouldn't be able to do LCC on a graph with 250M edges without at least 62 GB of cluster memory). However, NSCALE can process the partitions in sequence on a single machine (for such one-pass analytics tasks) by loading them one by one, thus the maximum memory needed

at any specific time point can be lower (at the expense of increased wall-clock time). On the other hand, for iterative tasks, NSCALE's limits mirror those of Giraph or GraphLab in that, there must be enough cluster memory to load all the partitions. Some of the recent graph processing systems like X-Stream and GraphChi do not have this restriction because of their use of disk-based processing; in future work, we plan to investigate how the NSCALE programming model may be adapted to such settings.

5.7 NSPARK: Porting NSCALE on Apache Spark

The NSCALE framework currently runs on the Apache Yarn platform as the underlying mechanism of distribution of data and computation. The GEP phase in NSCALE is based on Apache Hadoop and has been implemented as a multistage Map-Reduce job (Ref Section 5.3.1). The NSCALE runtime is distributed as a library on the cluster and runs as an embedded process inside the reducers of the final stage Map-Reduce job. As has been seen in the experimental evaluation, NSCALE provides great benefits in terms of scalability, performance and an intuitive API as compared to other existing vertex-centric approaches.

Having successfully built and evaluated NSCALE over the Apache YARN-Hadoop framework, we further explored the feasibility of porting our generalized framework for large-scale graph analytics over other big data platforms. Apache Spark [43] has emerged as a popular big data analytics platform in the recent past. It provides a unique ability to prune large datasets through a series of coarse-grained transformations and hold them in distributed memory for further analysis providing great performance benefits especially

for iterative analysis tasks. It uses a lineage graph to achieve fault tolerance without resorting to intermediate state materialization as is done in Hadoop MapReduce. As such, it is a viable platform for big data data analytics which provides transparent distribution of data and computation and fault tolerance at scale.

5.7.1 Challenges Involved

As we have seen in our experimental evaluations, GraphX, a graph analytics library that sits atop the Spark platform, does not scale well for large-scale graphs. It emulates the vertex-centric programming frameworks thus suffering from the same limitations. The storing of the vertex and edge information as separate immutable RDDs further aggravates the problem of aggregating neighborhood state and executing user computation on it. We therefore need a system that can accrue the benefits of fault-tolerant in-memory computation on large distributed data sets as provided by Spark while providing a scalable solution for complex subgraph-centric graph analysis tasks. The unique set of challenges that required to be addressed for porting NSCALE on Spark can be summarized as follows:

- Providing an efficient mechanism for extracting the user-defined subgraphs from the underlying raw graph data using a series of coarse-grained transformations supported by the Spark API.
- Designing and developing an appropriate abstraction for holding the extracted subgraphs in a distributed setting while minimizing the memory footprint for the same.
- Designing an intuitive API for enabling users to specify the subgraphs of interest and the user computation that runs on these subgraphs.

- Building an efficient execution model that would execute the user computation on the extracted subgraphs in distributed memory without incurring the overheads of the vertex-centric approaches.

- Providing support for both one pass and iterative analytics while keeping in mind the limitations of the Spark execution model arising due to the immutability of the RDDs.

5.7.2 Our Approach

We followed a three-phase approach for designing and building an initial prototype for NSPARK. We explain the details of the NSPARK architecture and functionality in these three phases discussed below:

Phase 1: Building the GEP phase in NSPARK. The NSCALE GEP phase has been implemented on Apache Spark using a series of RDD transformations in Scala. We describe the detailed steps below:

- Starting from a raw edge list representation of the underlying graph data we have designed and built a series of coarse-grained transformations that construct and extract the relevant subgraphs of interest instantiated in memory as a Spark RDD.

- These subgraphs are then provided as input to the shingle-based bin-packing algorithm that we discussed earlier in this chapter (Ref Section 5.3.2). The bin packing algorithm has been ported to Scala and groups together subgraphs based on neighborhood similarity to minimize the memory foot print of the subgraphs held in

distributed memory. The final output of the bin packing algorithm is a query-vertex (or subgraph)-to-partition mapping.

- Once this mapping information is obtained, it is then joined with the subgraph structural information that we had extracted earlier to produce a memory efficient distributed instantiation of the extracted subgraphs. We discuss the details of that in phase 2 below.

Phase 2: Instantiating the subgraphs in distributed memory. We had built a graph library in NSCALE that provided the data structures for holding the subgraphs in memory as well as the bitmap implementations required for the distributed and parallel execution of user computation. The graph library exported the popular BluePrints API, a generic API, that binds to a large number of graph database backends (e.g., Neo4j) and is used by many open source graph processing frameworks. Implementing the BluePrints API thus enables the use of existing toolkits and programs over large graphs. To carry forward this advantage we ported the same graph library to NSPARK. Once the BluePrints-based graph library was available within the NSPARK environment we used the following steps to instantiate the subgraphs of interest in distributed memory.

- The subgraph structural information extracted in Phase 1 was joined with the subgraph-to-partition information obtained from the bin packing algorithm within a coarse-grained map transformation. This enabled us to group the subgraphs using the partition number as the grouping key.

- The graph library API was used within the transformation to construct a graph

object for each partition. Each graph object now contains a set of subgraphs that had been binned together into a partition by the bin packing algorithm.

- The final output of this phase was a Spark RDD containing a set of BluePrint graph objects as described above, ready for executing user computation by the execution engine.

Phase 3: Executing user computation. We have ported the NSCALE execution engine written in Java to NSPARK with some modifications. These modifications include some design changes to the Master-worker architecture of the execution engine which enable it run within the Spark coarse grained transformations, take the RDD graph objects as input for executing user computation and provide an output RDD to store the results of user computation. We explain the details of the execution phase below:

- We use the graph object Spark RDD obtained from phase 2 as input and apply a map transformation. The execution engine is instantiated within the map transformation creating a separate instance for each graph object within the RDD. This design choice seamlessly enables us to use the Spark platform functionality to create an instantiation of a distributed execution engine.

- Within each instantiation of the execution engine the Master process of the execution engine spawns several worker threads within a thread pool whose size is governed by the underlying hardware of the machine running the Spark executor instance.

- The worker threads execute the compute function written by the user (using the

BluePrints API) on the subgraphs within each graph object of the RDD. The bitmap implementation provided by our library controls the scope of computation for each worker thread while enabling the parallel execution of user computation on subgraphs that have been stored in an overlapped fashion in memory.

- The design choice of using the bit map based NSCALE execution engine within the NSPARK framework thus enables the distributed execution of user computation while minimizing memory consumption by exploiting overlap among the neighborhoods of interest.

- The NSPARK design draws from a unique performance advantage of the underlying Spark platform. Once an RDD is created, it can be persisted in memory and be repeatedly used for different analysis tasks. The NSPARK design benefits from this wherein the graph RDD object created by the GEP phase can be persisted and used as input for several graph analytics tasks thus amortizing the cost of GEP phase across different analytics tasks. This also gives us the ability to meaningfully compose more complex tasks as chains wherein the output of a previous task can be directly fed as input to the next tasks in the chain.

5.7.3 Experimental Evaluation

We have tested the NSPARK prototype system using two real world data sets and four applications that had been used to evaluate NSCALE. Fig 5.17 shows the results for the performance comparisons of NSPARK with NSCALE. Comparing the performance of NSPARK with NSCALE in terms of computational effort (CE- node secs) for two differ-

ent data sets Web NotreDame and Web Google (Ref Figures 5.17(a), 5.17(b)), we see that NSPARK performs a little better which can be attributed to a better performance of the GEP phase on the Spark platform as compared to a multistage map reduce implementation in hadoop. As far as the memory consumption is concerned (Ref Figures 5.17(c), 5.17(d)), NSPARK consumes a little more memory than NSPARK. We used a single executor instance for our experiments with 35GB of memory and 15GB of driver memory. The maximum virtual memory actually used by the Spark instance was 25.3GB across all experimental runs.

Figure 5.17(e) shows the performance breakdown of the different components of NSPARK in terms of the computational effort. We see that similar to NSCALE the user computation is still the major part of the computational effort as compared to the GEP phase. GEP phase in NSPARK has been further broken down into subgraph extraction, bin packing and the actual construction of the graph RDD object. Finally we compare the performance of NSPARK with NSCALE and GraphX. We see that our implementation of NSPARK performs much better than GraphX which can be attributed to a better design of the abstractions that hold the graph in memory and a better execution model which can take advantage of overlapped execution.

We have evaluated our NSPARK prototype on the Apache Spark platform. In this prototype we focus on one pass analytics to ascertain the feasibility and functionality of the port. Support for iterative applications can be added on similar lines as NSCALE while using techniques that minimize the memory footprints keeping the immutability of the RDDs in mind. Since the extension to iterative applications is not a fundamental limitation of the NSPARK design we leave this extension as a future exercise.

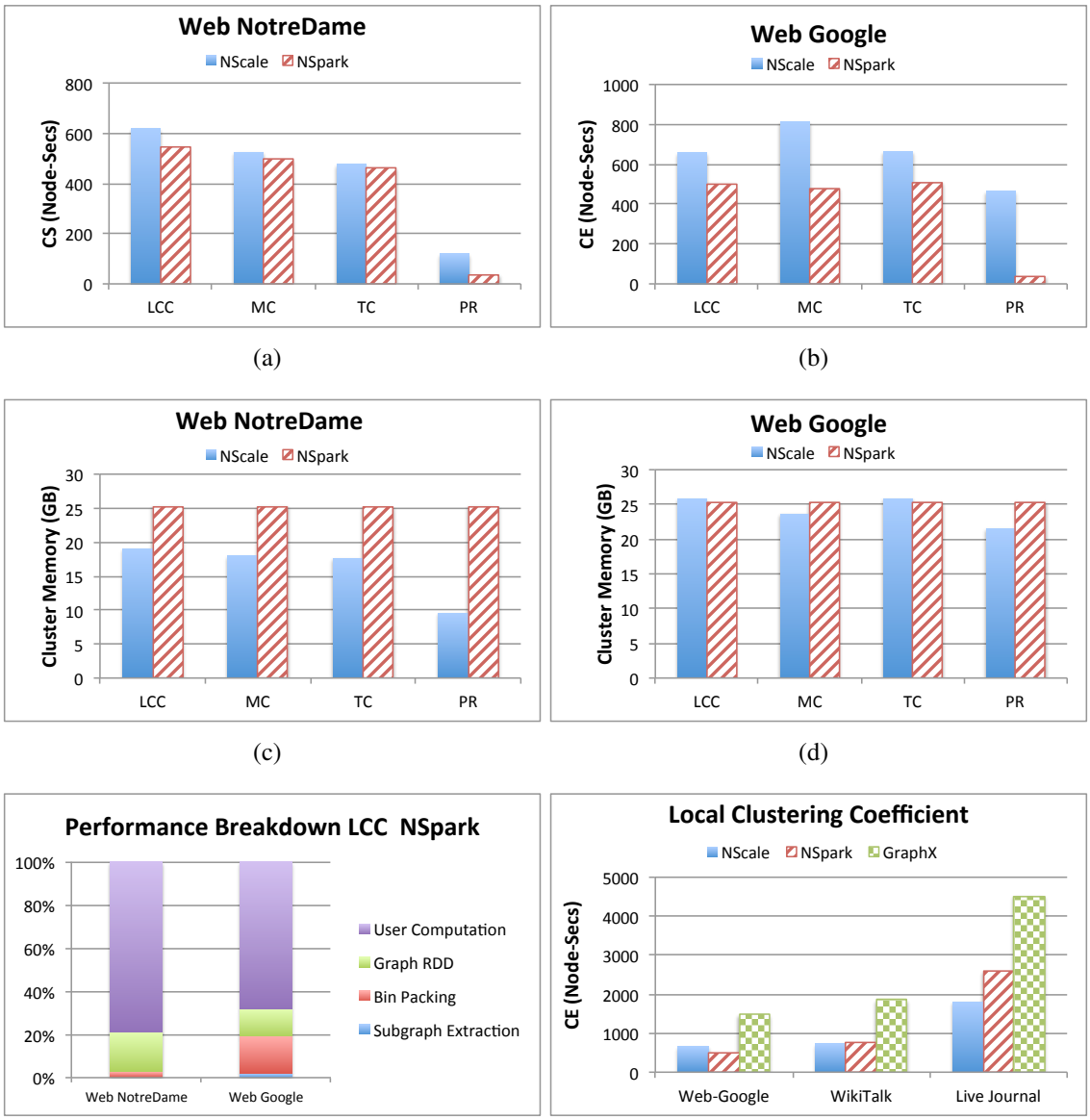


Figure 5.17: **NSPARK Performance : (a-b) Computational Effort (CE (node-secs)) comparison with NSCALE; (c-d) Cluster memory (GB) comparison with NSCALE; (e) Performance breakdown of NSPARK. (f) Performance comparison with NSCALE and GraphX.**

5.8 Conclusion

Increasing interest in performing graph analytics over very large volumes of graph data has led to much work on developing distributed graph processing frameworks in recent years, with the vertex-centric frameworks being the most popular. Those frameworks are, however, severely limited in their ability to express and/or efficiently execute complex and rich graph analytics tasks that network analysts want to pose. We argue that both for ease-of-use and efficiency, a more natural abstraction is a *subgraph-centric framework*, where the users can write computations against entire subgraphs or multi-hop neighborhoods in the graph. We show how this abstraction generalized the vertex-centric programming framework, how it is a natural fit for many commonly used graph analytics tasks, and how it leads to more efficient execution by reducing the communication and memory overheads. We also argue that the graph extraction and loading phase should be carefully optimized to reduce the number of machines required to execute a graph analytics task, because of the non-linear relationship between that parameter and the total execution cost; we developed a novel framework for solving this problem, and we show that it can lead to significant savings in total execution time. Our comprehensive experimental evaluation illustrates the ability of our framework to execute a variety of graph analytics tasks on very large graphs, when Apache Giraph, GraphLab and GraphX fail to execute them on relatively small graphs.

Chapter 6: Conclusion and Future Directions

6.1 Conclusion

This dissertation focuses on building cost-effective cloud-based big data management systems. We demonstrate that our techniques affect a substantial reduction in the cost of data processing in the cloud in a variety of application domains for different real world datasets.

The first part of the dissertation focusses on reducing the cost of transactional workloads using workload-aware data placement and replication techniques. We model the workload as a hyyergraph and show that our compression technique enables us to scale to large workloads and database sizes. Our in-graph replication mechanism enables us to further reduce the number of distributed read transactions while still providing a user-defined level of availability for all data items. We provide an incremental solution to deal with changes in workload while minimizing the amount of data migration as compared to a complete repartitioning of the workload. We have built an effective routing mechanism that can take full advantage of the workload-aware data placement while efficiently dispatching transactions to appropriate partitions.

Our experimental evaluation of SWORD deployed on an Amazon EC2 cluster demonstrates that our hypergraph-based workload representation and use of in-graph

replication based on access patterns, lead to a much better quality data placement as compared to other data placement techniques. We show that our scaling techniques result in orders-of-magnitude reductions in the partitioning overheads including the workload partitioning time, cost of distributed transactions, and query routing times for data sets consisting of up to a billion tuples. Our incremental repartitioning technique effectively deals with the performance degradation caused by workload changes using minimal data movement. We also show that our techniques provide graceful tolerance to partition failures compared to other data placement techniques.

The second part of the dissertation describes how *resource consolidation* and *progressive analytics* can be used as effective means of reducing the cost of analytics over large volumes of data in the cloud. Specifically, we design, build and evaluate two systems: NOW! and NSCALE, to validate the effectiveness of these techniques in two different application domains.

NOW! is a progressive analytics system for large-scale data in the relational domain. NOW! allows users to communicate progressive samples to the system and provides platform support for efficient and deterministic query processing over these samples. Further, it provides repeatable semantics and provenance to data scientists. The progressive computation in the system is realized using an unmodified temporal streaming engine, by reinterpreting the temporal event fields as progress. NOW! has been built as a progressive data-parallel computation framework for the cloud which provides support for progressive SQL over big data on Azure. Our large-scale experiments show orders of magnitude performance gains affecting substantial reduction in the cost of analytics in the cloud.

NSCALE is a framework for subgraph-centric data analytics on large-scale graph

structured data in the cloud. The core contributions of this work are: (1) A subgraph-centric programming model that allows users to write programs against and declaratively specify subgraphs of interest; (2) Platform support for extracting the subgraphs of interest from the underlying raw graph data; (3) Use of a cost-based optimizer to pack the extracted subgraphs of interest in as few containers (i.e. memory) as possible to minimize resource allocation making the framework amenable to execution in cloud environments and (4) A distributed execution engine which uses novel techniques to reduce memory footprint by utilizing the overlap among the subgraphs of interest. Our comprehensive experimental evaluation of NSCALE against the state-of-the-art graph analytics systems such as Apache Giraph, GraphLab, GraphX for a variety of applications and real world large-scale datasets, illustrates the scalability and efficiency of our framework as compared to these frameworks.

6.2 Limitations

All the above mentioned systems have been built and extensively evaluated as prototypes. We now mention some of the current limitations of these systems:

- **SWORD.** The workload-aware data placement mechanism in SWORD has a few limitations. First, it does not take into account the heterogeneity of the machines on which the data is partitioned. Optimizing data placement in presence of heterogeneity is a current area of research and could lead to reduction of straggler effects and further improvement in performance for transactional workloads. Second, the routing mechanism requires set cover computation for each incoming query in or-

der to direct it to the appropriate partitions. Since this is an expensive computation an incremental approach that reuses work across different set cover computations could further improve the routing efficiency of the system. We leave both these optimizations for further enhancing performance as future work.

- **NOW!.** We have extensively tested the system wherein we resort to complete in-memory data flow and analytics which requires failures to cascade back to source data. Although restarting a job on failure is a cheap and practical solution, HA with low recovery time is a desirable feature in some cases. Since our primary focus was on supporting the progressive model over large-scale relational data, we have proposed several techniques for providing high availability (HA) and resilience to failures in NOW! and left the evaluation of such fine-grained HA using existing techniques that apply to our setting, as future work.

- **NSCALE** As our comprehensive experimental evaluation shows, NSCALE is able to scale better than the other systems we compared against for most of the tasks, and it uses fewer resources in general. There is certainly a limit to the graph sizes that our current implementation can handle given limited resources, and those can be seen or extrapolated from our reported numbers (e.g., NSCALE wouldn't be able to do LCC on a graph with 250M edges without at least 62 GB of cluster memory). However, NSCALE can process the partitions in sequence on a single machine (for such one-pass analytics tasks) by loading them one by one, thus the maximum memory needed at any specific time point can be lower (at the expense of increased wall-clock time). For iterative tasks, the limits of NSCALE should

mirror those of Giraph or GraphLab since the NSCALE framework is equivalent to those vertex-centric frameworks for such tasks. In other words, the graph must fit in distributed memory (XStream and GraphChi do not have this restriction because of their use of disk-based processing) and the total number of iterations should be reasonable (to keep the network communication low).

6.3 Future Directions

In the near future we envision a dominance of cloud-based storage and compute systems due to the economies of scale. These systems would provide a plethora of services that would be used in an environment that is characterized by the ubiquity of mobile devices and platforms that are becoming ever more powerful. Anytime, anywhere services that transcend geographic boundaries bring with them a host of challenges for distributed data storage, interactive querying and analysis of ever increasing volumes of data. Some of these include building data management systems for the next generation that support the real-time ingest and analysis of large volumes of streaming data to derive both deep insights such as business intelligence, climate change and also actionable results in real time for applications such as short term weather prediction, anomaly detection, cyber attack prevention, financial trade, etc.

We believe that SWORD, NOW! and NSCALE are a step in the right direction towards designing systems that enable resource consolidation in a cloud computing environment and providing efficient and cost effective solutions for data querying and analysis on large-scale data in the cloud. With the above mentioned scenario in mind we discuss

below a broad set of future directions for the work that has been presented in this dissertation:

6.3.1 Multi-tenancy and Workload Consolidation

In this dissertation we have considered workload-aware data partitioning and replication for transactional workloads for providing Database-as-a-Service in cloud computing environments. This work focusses on workloads that run against a database service in a single tenant environment in the cloud. However, as we are aware the cloud can be a multi-tenant environment and therefore the next step in the direction would be to provide support for consolidating the workload to affect a smarter data placement in an environment that supports multi-tenancy.

Workload-aware data placement in a multi-tenant environment has its own sets of challenges for the cloud service provider. These include meeting the SLAs for different client workloads while minimizing the costs accrued for utilizing the underlying cloud infrastructure. This would require us to explore methods of modeling multiple workloads from different clients each with its own characteristic access patterns and service requirements. Modeling this as a consolidated workload while meeting the individual requirements of individual workloads to affect an appropriate data placement that minimizes the cost accrued by the service provider while doing this at scale would be a challenging problem to address.

6.3.2 Progressive Analytics in the Graph Analytics Domain

We have shown the benefits of progressive analytics over big data in the relational domain in this dissertation. An interesting direction for future work would be to explore ways of bringing the same benefits to the graph analytics domain. We have taken some initial steps towards building a simple prototype system for progressive analytics on large-scale graphs that enables users to execute an analysis task on user-defined progressive samples of graph data to produce early results.

There are a number of challenges that would need to addressed while building such a system, some which are: (1) enabling users to choose/encode a custom sampling strategy depending on the analysis task, (2) developing and building platform support for an algebra for progressive graph operators to enable users to compose meaningful progressive analysis tasks, and (3) providing support for appropriate execution models (vertex-centric, subgraph-centric, sample-centric, etc.) for executing user computation on the samples in distributed memory producing progressive results. We believe that such a system would be a step in the right direction in addressing the challenges associated with reducing the cost of data analytics on large-scale graph structured data in the cloud.

6.3.3 Addressing Disruptions from Hardware Improvements

We envision disruptions coming from rapid improvements of hardware technologies that would bring significant improvement to storage access costs, caching mechanisms, network throughput and latencies and the compute capabilities of mobile devices. Together, these improvements would open up exciting new research challenges and ne-

cessitate revisiting the design of query optimizers, schedulers, data flow, placement and communication mechanisms in large-scale data analysis platforms and querying engines in a distributed environment. We believe that addressing these challenges would be essential in maintaining the viability of such large-scale data management systems in presence of the improvements in the underlying hardware that these systems have been built and deployed on.

Bibliography

[1] D. J. Dewitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 1992.

[2] J. Gray and L. Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 2003.

[3] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[4] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 2008.

[5] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[6] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, September 2010.

[7] D. Cohn, L. Atlas, and R. Ladner. Improving Generalization with Active Learning. *Mach. Learn.*, 15, 1994.

[8] M. D. McKay, R. J. Beckman, and W. J. Conover. Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21, 1979.

[9] O. Maron and A. W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *NIPS*, 1993.

[10] J. M. Hellerstein and R. Avnur. Informix under control: Online query processing. *Data Mining and Knowledge Discovery Journal*, 2000.

[11] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. SIGMOD '07.

[12] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[13] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD*, 2004.

[14] A. Doucet, M. Briers, and S. Senecal. Efficient block sampling strategies for sequential monte carlo methods. *Journal of Computational and Graphical Statistics*, 2006.

[15] P. J. Haas and J. M. Hellerstein. Join algorithms for online aggregation. In *IBM Research Report RJ 10126*, 1998.

[16] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. VLDB '99.

[17] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[18] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: Scalable Workload-aware Data Placement for Transactional Workloads. In *EDBT*, 2013.

[19] K. A. Kumar, A. Quamar, A. Deshpande, and S. Khuller. SWORD: workload-aware data placement and replica selection for cloud data management systems. *VLDB J.*, 23(6):845–870, 2014.

[20] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Large-Scale Graph Analytics in the Cloud. *CoRR*, abs/1405.1499, 2014.

[21] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Analytics on Large Graphs. *PVLDB*, 7(13):1673–1676, 2014.

[22] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Large-Scale Graph Analytics in the Cloud. *VLDB J.*, 2015.

[23] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup Tables: Fine-Grained Partitioning for Distributed Databases. In *ICDE*, 2011.

[24] N. Bruno, S. Chaudhuri, A. C. Konig, V. R. Narasayya, R. Ramamurthy, and M. Syamala. Autoadmin project at Microsoft Research: Lessons learned. *IEEE Data Eng. Bull.*, 2011.

[25] K. A. Kumar, A. Deshpande, and S. Khuller. Data placement and replica selection for improving colocation in distributed environments. CoRR, 2012.

[26] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[27] A. Lakshman and P. Malik. Cassandra: Structured storage system on a P2P network. In *PODC '09*.

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06*.

[29] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, 2011.

[30] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[31] B. Kemme and A. Gustavo. Database replication: a tale of research across communities. *PVLDB*, September 2010.

[32] R. J. Peris, M. P. Martnez, B. Kemme, and G. Alonso. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. *IEEE Symposium on RDS*, 2001.

[33] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, 1996.

[34] J. R. Peris, M. P. Martinez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM TODS*, 28(3) 2003.

[35] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 2011.

[36] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *PVLDB 2012*.

[37] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

[38] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable progressive analytics on big data in the cloud. Technical report, MSR. `http://aka.ms/Jpe5f5`.

[39] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD 2011*.

[40] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[41] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB 2010*.

[42] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 2012.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI'12.

[44] M. Zaharia, T. Das, H. Li, Hunter T., S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[45] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD, 2010*.

[46] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 2012.

[47] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking pulse of a fast-changing and connected world. In *EuroSys*, 2012.

[48] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[49] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, 2013.

[50] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. In *NIPS*, 2012.

[51] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 2013.

[52] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB*, 2013.

[53] Y. Simmhan, A. G. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. *CoRR*, 2013.

[54] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.

[55] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[56] I. Hoque and I. Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *TRIOS*, 2013.

[57] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A System for Searching the Social Graph. *Proc. VLDB Endow.*, 2013.

[58] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.

[59] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[60] hMetis: a hypergraph partitioning package, http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[61] C. Ayka, B.Cambazoglu, and U. Bora. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 2008.

[62] X. Wang, A. Smalter, J. Huan, and G. H. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In *EDBT*, 2009.

[63] R. J. Peris and M. P. Martinez. How to select a replication protocol according to scalability, availability and communication overhead. In *SRDS*, 2001.

[64] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *SIGMOD*, 1999.

[65] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999*.

[66] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable progressive analytics on big data in the cloud. *PVLDB*, 2013.

[67] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP Server and Online Behavioral Targeting. In *VLDB*, 2009.

[68] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Y. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A Query Processing Engine for Data Streams. In *ICDE*, 2004.

[69] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[70] C. Jensen and R. Snodgrass. Temporal specialization. In *ICDE*, 1992.

[71] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.

[72] E. Ryvkina, A. Maskey, M. Cherniack, and S. B. Zdonik. Revision processing in a stream processing engine: A high-level design. In *ICDE*, 2006.

[73] B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. In *ICDE*, 2012.

[74] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI, 2004*.

[75] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD*, 2011.

[76] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *HotCDP*, 2012.

[77] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[78] T. White. *Hadoop: The Definitive Guide*. 2009.

[79] The LINQ Project. `http://aka.ms/rjhi00`.

[80] Daytona for Azure. `http://aka.ms/unkcbq`.

[81] R. Barga, J. Ekanayake, and W. Lu. Iterative mapreduce research on Azure. In *SC*, 2011.

[82] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 2002.

[83] L Backstrom and J Leskovec. Supervised random walks: Predicting and recommending links in social networks. In *WSDM*, 2011.

[84] L. Akoglu, M. McGlohon, and C. Faloutsos. OddBall: spotting anomalies in weighted graphs. In *PAKDD*, 2010.

[85] Apache Giraph http://giraph.apache.org.

[86] J. Huang., D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. In *PVLDB*, 2011.

[87] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 2004.

[88] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *WWW*, 2013.

[89] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[90] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD*, 2004.

[91] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[92] P. Zhao, J. X. Yu, and P. S. Yu. Graph Indexing: Tree + Delta less than equal to Graph. In *VLDB*, 2007.

[93] L Zou, L Chen, J. X. Yu, and Y. Lu. A Novel Spectral Coding in a Large Graph Database. In *EDBT*, 2008.

[94] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *J. ACM*, 1976.

[95] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2004.

[96] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *VLDB*, 2008.

[97] H. He and A. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *SIGMOD*, 2008.

[98] Y. Tian and J. M. Patel. TALE: A Tool for Approximate Large Graph Matching. In *ICDE*, 2008.

[99] M. Mongiov, R. D. Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. Sigma: a set-cover-based inexact graph matching algorithm. *J. Bioinformatics and Computational Biology*, 2010.

[100] J. Lee, W. S. Han, R. Kasperovics, and J. H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. In *PVLDB*, 2013.

[101] BluePrints API: https://github.com/tinkerpop/blueprints/wiki.

[102] Gremlin: http://github.com/tinkerpop/gremlin/wiki.

[103] Furnace: https://github.com/tinkerpop/furnace/wiki.

[104] J. M. Pujol, V. Erramilli, G. Siganos, Xiaoyuan Y. 0001, N. Laoutaris, P. Chhabra, and P. Rodriguez. The Little Engine(s) That Could: Scaling Online Social Networks. In *SIGCOMM*, 2010.

[105] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. 2011.

[106] T. Izumi, T. Yokomaru, A. Takahashi, and Y. Kajitani. Computational complexity analysis of set-bin-packing problem. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 81(5):842–849, 1998.

[107] A Rajaraman and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[108] Metis: http://glaros.dtc.umn.edu/gkhome/metis.

[109] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *SIGKDD*, 2006.

[110] A. Daniel Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics. *Proc. VLDB Endow.*, 2013.

[111] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*

[112] Redis http://redis.io/.

[113] Stanford Network Analysis Project: https://snap.stanford.edu.

[114] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics*, 2012.

[115] M. S. Granovetter. The strength of weak ties. *American Journal of Sociology*, 1973.

[116] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB, 2012*.