Title
On Software Infrastructure for Scalable Graph Analytics

Permalink
https://escholarship.org/uc/item/7px2636z

Author
Bu, Yingyi

Publication Date
2015

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

On Software Infrastructure for Scalable Graph Analytics

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Yingyi Bu

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Michael T. Goodrich
Professor Tyson Condie

2015

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Professor Michael J. Carey, for supporting me throughout the past five years. Professor Carey is a serious system researcher, a knowledgeable data management veteran, a smart, critical thinker, a patient teacher, and a humorous supervisor. His deep knowledge and insightful vision of the area guards me to work on the right direction, solve the right problem, build the right system, and do the right experiments. Professor Carey's "BMW" (build, measure, write) research approach fundamentally changed my mindset and my way of doing research, and shaped me to be an independent researcher and a serious system builder. I will never forget his carefulness on every issue and every user inquiry of the systems that we have been building, e.g., AsterixDB, Hyracks, and Pregelix. I will never forget his enthusiasm in every potential performance or usability improvement of the systems that we have been building. I will never forget his patience in educating academic kids like me. Eventually all his efforts make my doctoral study in UC Irvine the most rewarding and memorable experience that I will cherish forever. Thanks, Professor Carey!

I would like to thank Professor Tyson Condie for joining my dissertation committee. Professor Condie has guided my research projects throughout my entire doctoral study and has provided me enormous suggestions on my work, from system architectures to implementations. He helped me tremendously on how to look at a system from a fundamental perspective and how to think of a research problem at a higher level. He spent a lot of time helping me improve the presentation of Chapter 3 of this thesis, from high-level pictures to low-level wordings.

I would like to thank Professor Michael T. Goodrich for joining my dissertation committee too. Professor Goodrich and his Ph.D. students were the early adopters of the Pregelix system, tried to implement real-world graph analysis problems on top of it, and provided us invaluable feedback on how the system could be improved for solving real-world graph problems.

I would like to thank Professor Chen Li and Professor Xiaohui Xie for developing the genome assembly application on top of Pregelix. Professor Li and Professor Xie, as well their students gave us many invaluable suggestions on how Pregelix could be improved for solving the genome assembly problem in a simple and efficient way.

I would like to thank Professor Harry Guoqing Xu for his tremendous help on Chapter 2 of this thesis, from formulating the research problem to writing up the paper. Professor Xu and his Ph.D. students continue working on the direction of the chapter and the exciting results they have obtained will greatly raise the impact of the chapter.

I would like to particularly thank Vinayak Borkar for his invaluable suggestions and discussions during my thesis work. Vinayak is not only a real system builder with tons of experiences but also an intelligent thinker. From him, I learned how to build a real system in a right way, how to design good abstractions, and how to analyze and improve the

performance. I will never forget the days and nights when we debugged and analyzed the performance of Pregelix and Hyracks together on a large cluster.

I would like to thank Jianfeng Jia for benchmarking Pregelix against distributed GraphLab and GraphX and thank Markus Holzemer for building the early prototype of the AsterixDB/Pregelix integration. I would like to thank Young-Seok Kim and Murtadha Hubail for guiding and reviewing my implementation of temporary dataset in AsterixDB. With the help from them, the quality of Chapter 3 and Chapter 4 is greatly improved.

I would like to thank all other AsterixDB team members and alumnus: Dr. Till Westmann, Dr. Nicola Onose, Dr. Alex Behm, Dr. Raman Grover, Dr. Sattam Alsubaiee, Dr. Rares Vernica, Pouria Pirzadeh, Ian Maxon, Abdullah Alamoudi, Inci Cetindil, Taewoo Kim, Zach Heilbron, Preston Carman, Ildar Absalyamov, Steven Jacobs, Madhusudan C.S, Khurram Faraaz, Kereno Ouaknine, Professor Vassilis Tsotras, Professor Heri Ramampiaro, and Professor Wenhai Li, for working hard together to make the systems stable and reliable. The results in this thesis stand on everyone's contributions here and there.

I would like to thank the following people who tried Pregelix at the early stage, reported bugs, hardened the system, and gave us feedback: Jacob Biesinger, Da Yan, Anbang Xu, Nan Zhang, Vishal Patel, Joe Simons, Nicholas Ceglia, Khanh Nguyen, Yuzhen Huang, Professor Hongzhi Wang, Professor James Cheng, Professor Chen Li, Professor Xiaohui Xie, Professor Harry Guoqing Xu, and Professor Michael T. Goodrich. Without their contributions, neither the system nor this thesis could not become real.

I would like thank Dr. Hongree Lee, Dr. Christopher Olston, Dr. Jayant Madhaven, Dr. Peter Hawkins, Dr. Vuk Ercegovac, and Dr. Alon Halevy for the wonderful mentoring during my summer internships at Google. From them, I learned Google's hybrid research approach that well aligns research visions with real-world products.

I would like to thank Google for providing me with a generous Google Ph.D. Fellowship during last two years of my Ph.D. program. The work reported in this thesis has also been supported by a UC Discovery grant, by NSF IIS awards 0910989, and by NSF CNS awards 1059436, 1305430, and 1305253. In addition, the AsterixDB project has benefited from generous industrial support from Amazon, eBay, Facebook, Google, HTC, InfoSys, Microsoft, Oracle Labs, and Yahoo.

I would like to thank my dear friends Zhijing Qin, Liyan Zhang, Wei Zhang, Zhi Chen, Yi Wang, Di Wu, Moulin Xiong, Zhenqiu Huang, Yongjie Zheng, Fang Deng, Jingwen Zhang, Yongxue Li, and Le An for helping and enriching my life in Irvine and in the future.

Finally, I would like to thank my parents Guilin and Jiling, my wife Yan, and my daughter Grace. Without their love and continuous encouragement, I could not make this thesis.

CURRICULUM VITAE

Yingyi Bu

EDUCATION

Doctor of Philosophy in Computer Science **2015**
University of California, Irvine *Irvine, California*

Master of Philosophy in Computer Science **2008**
The Chinese University of Hong Kong *Hong Kong, China*

Bachelor of Science in Computer Science **2005**
Nanjing University *Nanjing, Jiangsu, China*

REFEREED JOURNAL PUBLICATIONS

Pregelix: Big(ger) Graph Analytics on A Dataflow Engine **2014**
Proceedings of the Very Large Database Endowment (PVLDB)

AsterixDB: A Scalable, Open Source BDMS **2014**
Proceedings of the Very Large Database Endowment (PVLDB)

Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees **2014**
Proceedings of the Very Large Database Endowment (PVLDB)

The HaLoop Approach to Large-Scale Iterative Data Analysis **2012**
The VLDB Journal (VLDBJ)

HaLoop: Efficient Iterative Data Processing on Large Clusters **2010**
Proceedings of the Very Large Database Endowment (PVLDB)

Privacy Preserving Serial Data Publishing By Role Composition **2008**
Proceedings of the Very Large Database Endowment (PVLDB)

REFEREED CONFERENCE PUBLICATIONS

Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages **August 2015**
The 2015 ACM SIGMOD/SIGOPS Symposium on Cloud Computing (SOCC)

Facade: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications **March 2015**
The 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

A Bloat-Aware Design for Big Data Applications **June 2013**
The 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM)

Efficient Anomaly Monitoring Over Moving Object Trajectory Streams **June 2009**
The 15th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)

WAT: Finding Top-K Discords in Time Series Database **April 2007**
The 2007 SIAM International Conference on Data Mining (SDM)

SOFTWARE

Pregelix `http://pregelix.ics.uci.edu`
an open source distributed graph processing system that is based on an iterative dataflow design that is better tuned to handle both in-memory and out-of-core workloads.

AsterixDB `http://asterixdb.ics.uci.edu`
an open source Big Data management system, with new technologies for ingesting, storing, managing, indexing, querying, analyzing, and subscribing intensive semi-structured data.

ABSTRACT OF THE DISSERTATION

On Software Infrastructure for Scalable Graph Analytics

By

Yingyi Bu

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Michael J. Carey, Chair

Recently, there is a growing need for distributed graph processing systems that are capable of gracefully scaling to very large datasets. In the mean time, in real-world applications, it is highly desirable to reduce the tedious, inefficient ETL (extract, transform, load) gap between tabular data processing systems and graph processing systems. Unfortunately, those challenges have not been easily met due to the intense memory pressure imposed by process-centric, message passing designs that many graph processing systems follow, as well as the separation of tabular data processing runtimes and graph processing runtimes.

In this thesis, we explore the application of programming techniques and algorithms from the database systems world to the problem of scalable graph analysis. We first propose a bloat-aware design paradigm towards the development of efficient and scalable Big Data applications in object-oriented, GC enabled languages and demonstrate that programming under this paradigm does not incur significant programming burden but obtains remarkable performance gains (e.g., 2.5×+).

Based on the design paradigm, we then build Pregelix, an open source distributed graph processing system which is based on an iterative dataflow design that is better tuned to handle both in-memory and out-of-core workloads. As such, Pregelix offers improved performance characteristics and scaling properties over current open source systems (e.g., we have seen up to 15× speedup compared to Apache Giraph and up to 35× speedup compared to distributed GraphLab).

Finally, we integrate Pregelix with the open source Big Data management system AsterixDB to offer users a mix of a vertex-oriented programming model and a declarative query language for richer forms of Big Graph analytics with reduced ETL pains.

Chapter 1

Introduction

Over the past decade, the increasing demands for data-driven business intelligence have led to the proliferation of large-scale, data-intensive applications that often have huge amounts of data (often at terabyte or petabyte scale) to process.

In particular, there are more and more demands today to process Big Graphs for applications in social networking (e.g., friend recommendations), the web (e.g., ranking pages), and human genome assembly (e.g., extracting gene sequences). Statistics show that Google has indexed about 50 billion web pages [23]; Facebook has about 1.44 billion monthly active users [22]; the de Bruijn graph of human genomes has 3 billion+ nodes and 10 billion+ edges [109]; the knowledge graph which has already been used by Google Search has 570 million objects and more than 18 billion facts [16].

Even with those large numbers, the sizes of Big Graphs are still likely being underestimated because many huge, real-world graphs are hidden in data that look irrelevant to graphs. In machine-generated logs from web servers, sensors, and wearable devices, there are usually causal relationships between different log lines [6] from which we can build a graph. In Twittter, the relationships of "reply" and "retweet" among billions of tweets (e.g., 200 billion per year [24]) are edges of a gigantic tweet graph. In Google Hangouts [7], huge volumes of chats and talks contain many, many objects and relationships in the form of text and voice that can also form a huge knowledge graph. And the list goes on...

Analyzing graphs at such large scales exceeds the capability of a single computer and requires distributed graph processing platforms that can run on large-scale commodity machine clusters. The required platforms should not only be efficient, scalable, robust, and reliable, but should also offer a simple yet expressive programming model with support for easy graph ETL (extract, transform, load) operations.

Motivated by real-world use cases, this thesis focuses on issues and challenges in the design and implementation of Big Graph analytics platforms and studies the following problems:

- **What design principles should we follow to build efficient and scalable Big Data platforms?** Because of today's rapid development cycles, Java Virtual Machine-based languages like Java and Scala have become "de facto" programming languages for building Big Data processing platforms [10, 12, 92, 81, 108, 36, 42, 29]. However, the overhead of automatic garbage collection provided by those languages is often oppressive for performance-critical Big Data applications. To address the problem, Chapter 2 proposes a bloat-aware design paradigm for the development of efficient and scalable Big Data applications in managed object-oriented languages. Our key observation is that *the best way to collect garbage is not to generate it in the first place*, i.e., to keep the number of created objects bounded or approximately bounded at runtime. Our experimental results show that this new design paradigm is extremely effective in improving performance — even for the moderate-size data sets processed, we have observed 2.5×+ performance gains, and the improvement grows substantially with the size of the data set.
- **How should we build a Big Graph analytics platform that always works with its best effort regardless of cluster memory constraints?** Most current open-source

distributed graph computation platforms [9, 11, 73] follow a process-centric, message-passing design, which results in a “work if memory is enough, otherwise fail” situation. This approach wastes a lot of human effort in order to resize clusters and restart graph computation jobs, and it prevents many budget-limited organizations from analyzing Big Graphs. To solve this problem, Chapter 3 explores an architectural alternative that expresses message-passing semantics as iterative dataflows that run on a general-purpose data-parallel runtime. Our key observation is that *we can build graph computation platforms using database-style query evaluation techniques that have stood the test of time for working in memory-constrained environments.* Based on this new architecture, we have built an open source distributed graph processing system called Pregelix. In our experiments, while other graph computation platforms fail jobs in under-resourced settings, Pregelix always works gracefully. In addition, we have seen that Pregelix offers improved performance characteristics and scaling properties over other current open source systems (e.g., up to 35× speedup).

- **How should we support integrated, simple analytical pipelines for implicit Big Graphs?** More and more application scenarios [13] require Big Graph computation platforms to be able to consume tabular data from a data warehouse and to save results back into the data warehouse as well. In this context, *Big Graph analytics are no longer only about graphs, but also involve complex pipelines such as graph extraction and post-processing.* Unfortunately, most existing solutions require data scientists to manually glue together tabular data processing systems and graph processing systems, and to manage data format transformations between them, which distracts the data scientists’ from the logical analytical task itself. To reduce this burden, we have coupled AsterixDB, an open source Big Data management system for ingesting and more richly querying semi-structured data, with our Pregelix system to support such data science use cases. To do so, we added a non-ACID storage option (i.e., temporary dataset) into AsterixDB, and we also added a graph computation job entrance capability to AQL, the query language of AsterixDB. Chapter 4 presents the design and implementation of the resulting coupled system. The integration not only improves the user experience for complex graph analytics, but also enables *interactive graph analytics* by leveraging the secondary indexing capability of AsterixDB. Our experimental evaluation results demonstrate that the resulting system has very good scaling properties.

In this thesis, the design paradigm proposed in Chapter 2 serves as a foundation for the system implementations in Chapter 3 and 4. Chapter 3 describes the internals of the Pregelix system itself, and Chapter 4 presents how Pregelix is integrated with AsterixDB. In addition to providing an integrated, data-centric platform for Big Data management and graph analytics — since "one man's data is another man's graph" — the resulting architecture can serve as one potential model for coupling Big Data management with other types of Big Data analytics as well. In particular, this approach enables a data scientist (or a team of data scientists) to do end-to-end analysis using a combination of paradigms (e.g., AQL and Pregel) while doing so with a minimum of "gluing pain".

Chapter 2

A Bloat-Aware Design for Big Data Applications

2.1 Overview

Modern computing has entered the era of Big Data. The massive amounts of information available on the Internet enable computer scientists, physicists, economists, mathematicians, political scientists, bio-informaticists, sociologists, and many others to discover interesting properties about people, things, and their interactions. Analyzing information from Twitter, Google, Facebook, Wikipedia, or the Human Genome Project requires the development of scalable platforms that can quickly process massive-scale data. Such frameworks often utilize large numbers of machines in a cluster or in the cloud to process data in a parallel manner. Typical data-processing frameworks include data-flow and message passing runtime systems. A data-flow system (such as MapReduce [51], Hadoop [10], Hyracks [42], Spark [108], or Storm [95]) uses distributed file system to store data and computes results by pushing data through a processing pipeline, while a message passing system (such as

Pregel [75] or Giraph [9]) often loads one partition of data per processing unit (machine, process, or thread) and sends/receives messages among different units to perform computations. High-level languages (such as Hive [92], Pig [81], FlumeJava [46], or AsterixDB [29]) are designed to describe data processing at a more abstract level.

An object-oriented programming language such as Java is often the developer's choice for implementing data-processing frameworks. In fact, the Java community has already been the home of many data-intensive computing infrastructures, such as Hadoop [10], Hyracks [42], Storm [95], and Giraph [9]. Spark [108] is written in Scala, but it relies on a Java Virtual Machine (JVM) to execute. Despite the many development benefits provided by Java, these applications commonly suffer from severe memory bloat—a situation where large amounts of memory are used to store information not strictly necessary for the execution—that can lead to significant performance degradation and reduced scalability.

Bloat in such applications stems primarily from a combination of the inefficient memory usage inherent in the run time of a managed language as well as the processing of huge volumes of data that can exacerbate the already-existing inefficiencies by orders of magnitude. As such, Big Data applications are much more vulnerable to runtime bloat than regular Java applications. As an interesting reference point, our experience shows that the latest (Indigo) release of the Eclipse framework with 16 large Java projects loaded can successfully run (without any noticeable lag) on a 2GB heap; however, a moderate-size application on Giraph [9][1] with 1GB input data can easily run out of memory on a 12 GB heap. Due to the increasing popularity of Big Data applications in modern computing, it is important to understand why these applications are so vulnerable, how they are affected by runtime bloat, and what changes should be made to the existing design and implementation principles in order to make them scalable.

[1] Note that the version of Giraph that we studied throughout this chapter is svn revision 1232166, a version at year 2012 when we were doing the work of this chapter. A later version of Giraph (described in Section 4.5.1) with substantially reduced object creations will be experimentally evaluated in Chapter 3.

In this chapter, we describe a study of memory bloat using two real-world Big Data applications: Hive [92] and Giraph [9], where Hive is a large-scale data warehouse software (Apache top-level project, powering Facebook's data analytics) built on top of Hadoop and Giraph is an Apache open source graph analytics framework initiated by Yahoo!. Our study shows that *freely creating objects (as encouraged by object-orientation) is the root cause of the performance bottleneck that prevents these applications from scaling up to large data sets.*

To gain a deep understanding of the bottleneck and how to effectively optimize it away, we break down the problem of excessive object creation into two different aspects, (1) what is the space overhead if all data items are represented by Java objects? and (2) given all these objects, what is the memory management (i.e., GC) costs in a typical Big Data application? These two questions are related, respectively, to the spatial impact and the temporal impact that object creation can have on performance and scalability.

On the one hand, each Java object has a fixed-size header space to store its type and the information necessary for garbage collection. What constitutes the space overhead is not just object headers; the other major component is from the pervasive use of object-oriented data structures that commonly have multiple layers of delegations. Such delegation patterns, while simplifying development tasks, can easily lead to wasteful memory space that stores *pointers* to form data structures, rather than *the actual data* needed for the forward execution. Based on a study reported in [79], the fraction of the actual data in an IBM application is only 13% of the total used space. This impact can be significantly magnified in a Big Data application that contains a huge number of (relatively small) data item objects. For such small objects, the space overhead cannot be easily amortized by the actual data content. The problem of inefficient memory usage becomes increasingly painful for highly-parallel data-processing systems because each thread consumes excessive memory resource, leading to increased I/O costs and reduced concurrency.

On the other hand, a typical tracing garbage collection (GC) algorithm periodically traverses the entire live object graph to identify and reclaim unreachable objects. For non-allocation-intensive applications, efficient garbage collection algorithms such as a generational GC can quickly mark reachable objects and reclaim memory from dead objects, causing negligible interruptions from the main execution threads. However, once the heap grows to be large (e.g., a few dozens of GBs) and most objects in the heap are live, a single GC call can become exceedingly longer. In addition, because the amount of used memory in a Big Data application is often close to the heap size, GC can be frequently triggered and would eventually become the major bottleneck that prevents the main threads from making satisfactory progress. We observe that in most Big Data applications, a huge number of objects (representing data to be processed in the same batch) often have the same lifetime, and hence it is highly unnecessary for the GC to traverse each individual object every time to determine whether or not it is reachable.

Switch back to an unmanaged language? Switching back to an unmanaged language such as C++ appears to be a reasonable choice. However, our experience with many Big Data applications (such as Hive, Pig, Jaql, Giraph, or Mahout) suggests that a Big Data application often exhibits clear distinction between a *control path* and a *data path*. The control path organizes tasks into the pipeline and performs optimizations while the data path represents and manipulates data. For example, in a typical Big Data application that runs on a shared-nothing cluster, there is often a driver at the client side that controls the data flow and there are multiple run-time data operators executing data processing algorithms on each machine. The execution of the driver in the control path does not touch any actual data. Only the execution of the data operators in the data path manipulates data items. While the data path creates most of the run-time objects, its development often takes a very small amount of coding effort, primarily because data processing algorithms (e.g., joining, grouping, sorting, etc.) can be easily shared and reused across applications.

One study we have performed on seven open source Big Data applications shows that the data flow path takes an overage 36.8% of the lines of source code but creates more than 90% of the objects during execution. Details of this study can be found in Section 2.4.3. Following the conventional object-oriented design for the control path is often unharmful; *it is the data path that needs a non-conventional design and an extremely careful implementation.* As the control path takes the majority of the development work, it is unnecessary to force developers to switch to an unmanaged language for the whole application where they have to face the (old) problems of low productivity, less community resource, manual memory management, and error-prone implementations.

Although the inefficient use of memory in an object-oriented language is a known problem and has been studied before (e.g., in [79]), there does not exist any systematic analysis of its impact on Big Data applications. In this chapter, we study this impact both analytically and empirically. We argue that the designers of a Big Data application should strictly follow the following principle: *the number of data objects in the system has to be bounded and cannot grow proportionally with the size of the data to be processed.* To achieve this, we propose a new design paradigm that advocates to merge small objects in the storage and access the merged objects using data processors. The idea is inspired from old memory management technique called *page-based record management*, which has been used widely to build database systems. We adopt the proposed design paradigm in our "build-from-scratch" general-purpose Big Data processing framework (called Hyracks) at the application (Java) level, without requiring the modification of the JVM or the Java compiler. We demonstrate, using both examples and experience, that writing programs using the proposed design paradigm does not create much burden for developers.

We have implemented several common data processing tasks by following both this new design paradigm and the conventional object-oriented design principle. Our experimental results demonstrate that the implementations following the new design can scale to much larger data sizes than those following the conventional design. We believe that this new design paradigm is valuable in guiding the future implementations of Big Data applications using managed object-oriented languages. The observations made in this chapter strongly call for novel optimization techniques targeting Big Data applications. For example, optimizer designers can develop automated compiler and/or runtime system support (e.g., within a JVM) to remove the identified inefficiency patterns in order to promote the use of object-oriented languages in developing Big Data applications. Furthermore, future development of benchmark suites should consider the inclusion of such applications to measure JVM performance.

Contributions of this chapter include:

- an analytical study on the memory usage of common Big Data processing tasks such as the graph link analysis and the relational join. We find that the excessive creation of objects to represent and process data items is the bottleneck that prevents Big Data applications from scaling up to large datasets (Section 2.2);
- a bloat-aware design paradigm for the development of highly-efficient Big Data applications; instead of building a new memory system to solve the memory issues from scratch, we propose two application-level optimizations, including (1) merging (inlining) a chunk of small data item objects with the same lifetime into few large objects (e.g., few byte arrays) and (2) manipulating data by directly accessing merged objects (i.e., at the binary level), in order to mitigate the observed memory bloat patterns (Section 2.3);
- a set of experimental results (Section 2.5) that demonstrate significant memory and time savings using the design. We report our experience of programming for real-world data-processing tasks in the Hyracks platform (Section 2.4). We compare the performance of several Big Data applications with and without using the proposed design; the experimental results show that our optimizations are extremely effective (Section 2.5).

2.2 Memory Analysis of Big Data Applications

In this section, we study two popular data-intensive applications, Giraph [9] and Hive [92], to investigate the impact of creating of Java objects to represent and process data on performance and scalability. Our analysis drills down to two fundamental problems, one in space and one in time: (1) large space consumed by object headers and object references, leading to low packing factor of the memory, and (2) massive amounts of objects and references, leading to poor GC performance. We analyze these two problems using examples from Giraph and Hive, respectively.

2.2.1 Low Packing Factor

In the Java runtime, each object requires a header space for type and memory management purposes. An additional space is needed by an array to store its length. For instance, in the Oracle 64-bit HotSpot JVM, the header spaces for a regular object and for an array take 8 and 12 bytes, respectively. In a typical Big Data application, the heap often contains many small objects (such as `Integers` representing record IDs), in which the overhead incurred by headers cannot be easily amortized by the actual data content. Space inefficiencies are exacerbated by the pervasive utilization of object-oriented data structures. These data structures often use multiple-level of delegations to achieve their functionality, leading to large space storing *pointers* instead of the actual data. In order to measure the space inefficiencies introduced by the use of objects, we employ a metric called *packing factor*, which is defined as the maximal amount of actual data that be accommodated into a fixed amount of memory. While a similar analysis [79] has been conducted to understand the health of Java collections, our analysis is specific to Big Data applications where a huge amount of data flow through a fixed amount memory in a batch-by-batch manner.

To analyze the packing factor for the heap of a typical Big Data application, we use the PageRank algorithm [83] (i.e., an application built on top of Giraph [9]) as a running example. PageRank is a link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively computing the weight of each vertex based on the weight of its inbound neighbors. This algorithm is widely used to rank web pages in search engines.

We ran PageRank on different open source Big Data computing systems, including Giraph [9], Spark [108], and Mahout [34], using a 6-rack, 180-machine research cluster. Each machine has 2 quad-core Intel Xeon E5420 processors and 16GB RAM. We used a 70GB web graph dataset that has a total of $1,413,511,393$ vertices. We found none of the three systems could successfully process this dataset. They all crashed with `java.lang.OutOfMemoryError`, even though the data partitioned for each machine (i.e., less than 500MB) should easily fit into its physical memory.

We found that many real-world developers experienced similar problems. For example, we saw a number of complaints on `OutOfMemoryError` from Giraph's user mailing list, and there were 13 bloat-related threads on Giraph's mailing list from January 2012 to September 2012[2].

In order to locate the bottleneck, we perform a quantitative analysis using PageRank. Giraph contains an example implementation of the PageRank algorithm. Part of its data representation implementation is shown below.

```
public abstract class EdgeListVertex<I extends WritableComparable,
        V extends Writable,
        E extends Writable, M extends Writable>
        extends MutableVertex<I, V, E, M> {
    private I vertexId = null;
```

[2] http://mail-archives.apache.org/mod_mbox/giraph-user/

```
    private V vertexValue = null;

    /** indices of its outgoing edges */
    private List<I> destEdgeIndexList;

    /** values of its outgoing edges */
    private List<E> destEdgeValueList;

    /** incoming messages from
        the previous iteration */
    private List<M> msgList;
    ......

    /** return the edge indices starting from 0 */
    public List<I> getEdegeIndexes(){
               ...
    }
}
```

Graphs handled in Giraph are labeled (i.e., both their vertices and edges are annotated with values) and their edges are directional. Class `EdgeListVertex` represents a graph vertex. Among its fields, `vertexId` and `vertexValue` store the ID and the value of the vertex, respectively. Field `destEdgeIndexList` and `destEdgeValueList` reference, respectively, a list of IDs and a list of values of its outgoing edges. `msgList` contains incoming messages sent to the vertex from the previous iteration. Figure 2.1 visualizes the Java object subgraph rooted at an `EdgeListVertex` object.

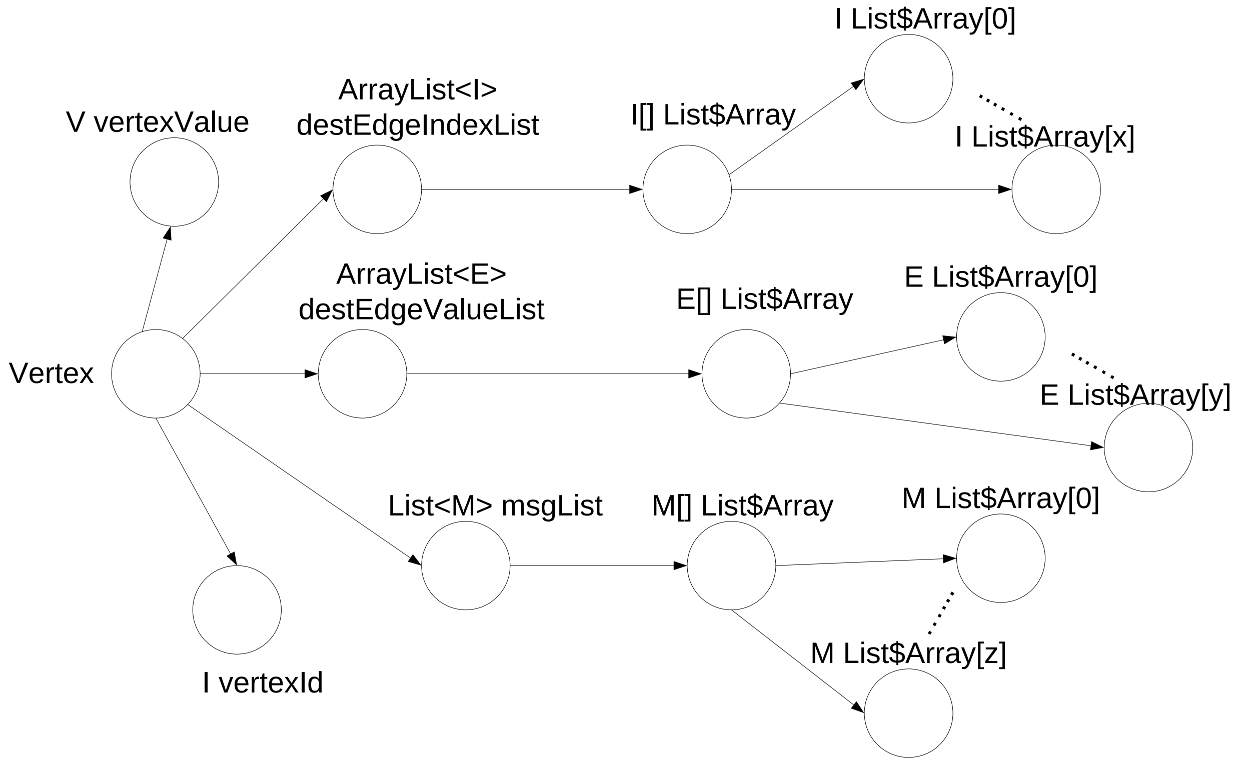


Figure 2.1: Giraph object subgraph rooted at a vertex.

In Giraph's PageRank implementation, the concrete types for I, V, E, and M are `LongWritable`, `DoubleWritable`, `FloatWritable`, and `DoubleWritable`, respectively. Each edge in the graph is equi-weighted, and thus the list referenced by `destEdgeValueList` is always empty. Assume that each vertex has an average of m outgoing edges and n incoming messages. Table 2.1 shows the memory consumption statistics of a vertex data structure in the Oracle 64-bit HotSpot JVM. Each row in the table reports a class name, the number of its objects

Class	**#Objects**	**Header (b)**	**Pointer (b)**
Vertex	1	8	40
List	3	24	24
List$Array	3	36	$8(m+n)$
LongWritable	$m+1$	$8m+8$	0
DoubleWritable	$n+1$	$8n+8$	0
Total	$m+n+9$	$8(m+n)+84$	$8(m+n)+64$

Table 2.1: Numbers of objects per vertex and their space overhead (in bytes) in PageRank in the Sun 64-bit Hopspot JVM.

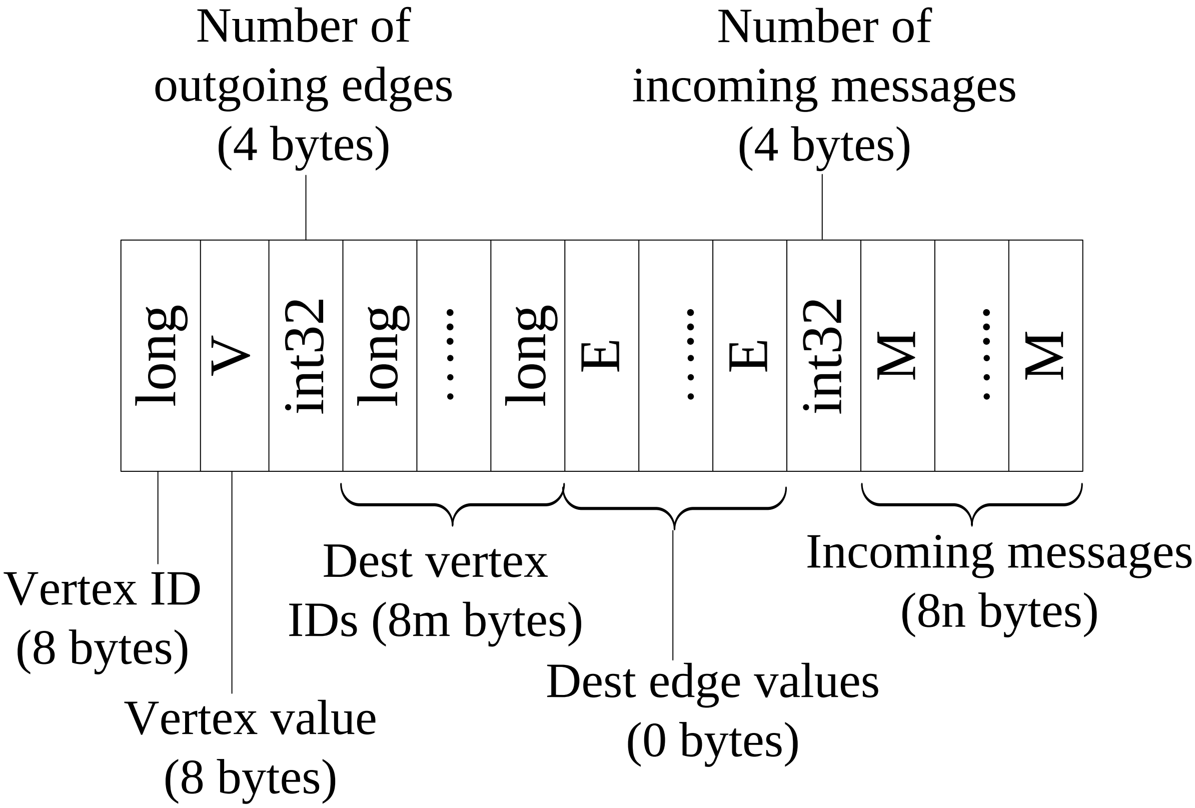


Figure 2.2: The compact layout of a vertex.

needed in this representation, the number of bytes used by the headers of these objects, and the number of bytes used by the reference-typed fields in these objects. It is easy to calculate that the space overhead for each vertex in the current implementation is $16(m + n) + 148$ (i.e., the sum of the header size and pointer size in Table 2.1).

On the contrary, Figure 2.2 shows an ideal memory layout that stores only the *necessary* information for each vertex (without using objects). In this case, the representation of a vertex requires $m + 1$ long values (for vertex IDs), n double values (for messages), and two 32-bit int values (for specifying the number of outgoing edges and the number of messages, respectively), which consume a total of $8(m+n+1)+16 = 8(m+n)+24$ bytes of memory. This memory consumption is even less than half of the space used for object headers and pointers in the object-based representation. Clearly, the space overhead of the object-based representation is greater than 200%.

2.2.2 Large Volumes of Objects and References

In a JVM, the GC threads periodically iterate all live objects in the heap to identify and reclaim dead objects. Suppose the number of live objects is n and the total number of edges in the object graph is e, the asymptotic computational complexity of a tracing garbage collection algorithm is $O(n+e)$. For a typical Big Data application, its object graph consists of a great number of isolated object subgraphs, each of which represents either a data item or a data structure created for processing items. As such, there often exists an extremely large number of in-memory data objects, and both n and e can be orders of magnitude larger than those of a regular Java application.

We use an exception example from Hive's user mailing list to analyze the problem. This exception was found in a discussion thread named "how to deal with Java heap space errors"[3]:

```
FATAL org.apache.hadoop.mapred.TaskTracker:
  Error running child : java.lang.OutOfMemoryError:
  Java heap space
   org.apache.hadoop.io.Text.setCapacity(Text.java:240)
  at org.apache.hadoop.io.Text.set(Text.java:204)
  at org.apache.hadoop.io.Text.set(Text.java:194)
  at org.apache.hadoop.io.Text.<init>(Text.java:86)
  ......
  at org.apache.hadoop.Hive.ql.exec.persistence.Row
    Container.next(RowContainer.java:263)
  org.apache.hadoop.Hive.ql.exec.persistence.Row
    Container.next(RowContainer.java:74)
  at org.apache.hadoop.Hive.ql.exec.CommonJoinOperator.
```

[3] http://mail-archives.apache.org/mod_mbox/Hive-user/201107.mbox/

```
    checkAndGenObject(CommonJoinOperator.java:823)
  at org.apache.hadoop.Hive.ql.exec.JoinOperator.
    endGroup(JoinOperator.java:263)
  at org.apache.hadoop.Hive.ql.exec.ExecReducer.
    reduce(ExecReducer.java:198)
  ......
  at org.apache.hadoop.Hive.ql.exec.persistence.Row
    Container.nextBlock(RowContainer.java:397)
  at org.apache.hadoop.mapred.Child.main(Child.java:170)
```

We inspected the source code of Hive and found that the top method `Text.setCapacity()` in the stack trace is not the cause of the problem. In Hive's join implementation, its `JoinOperator` holds all the `Row` objects from one of the input branches in the `RowContainer`. In cases where a large number of `Row` objects is stored in the `RowContainer`, a single GC run can become very expensive. For the reported stack trace, the total size of the `Row` objects exceeds the heap upper bound, which causes the `OutOfMemory` error.

Even in cases where no `OutOfMemory` error is triggered, the large number of `Row` objects can still cause severe performance degradation. Suppose the number of `Row` objects in the `RowContainer` is n. Hence, the GC time for traversing the internal structure of the `RowContainer` object is at least O(n). For Hive, n grows proportionally with the size of the input data, which can easily drive up the GC overhead substantially. The following is an example obtained from a user report at StackOverflow[4]. Although this problem has a different manifestation, the root cause is the same.

"I have a Hive query which is selecting about 30 columns and around 400,000 records and inserting them into another table. I have one join in my SQL clause, which is just an inner join. The query fails because of a Java GC overhead limit exceeded."

[4] http://stackoverflow.com/questions/11387543/performance-tuning-a-Hive-query.

In fact, complaints about large GC overhead can be commonly seen on either Hive's mailing list or the StackOverflow website. What makes the problem even worse is that there is not much that can be done from the developer's side to optimize the application, because the inefficiencies are inherent in the design of Hive. All the data processing-related interfaces in Hive require the use of Java objects to represent data items. To manipulate data contained in `Row`, for example, we have to wrap it into a `Row` object, as designated by the interfaces. If we wish to completely solve this performance problem, we would have to re-design and re-implement all the related interfaces from scratch, a task that any user could not afford to do. This example motivates us to look for solutions at the design level, so that we would not be limited by the many (conventional) object-oriented guidelines and principles.

2.3 The Bloat-Aware Design Paradigm

The fundamental reason for the performance problems discussed in Section 2.2 is that the two Big Data applications were designed and implemented the same way as regular object-oriented applications: *everything is object*. Objects are used to represent both *data processors* and *data items* to be processed. While creating objects to represent data processors may not have significant impact on performance, the use of objects to represent data items creates a big scalability bottleneck that prevents the application from processing large data sets. Since a typical Big Data application does similar data processing tasks repeatedly, a group of related data items often has similar liveness behaviors. They can be easily managed together in large chunks of buffers, so that the GC does not have to traverse each individual object to test its reachability. For instance, all vertex objects in the Giraph example have the same lifetimes; so do `Row` objects in the Hive example. A natural idea is to allocate them in the same memory region, which is reclaimed as a whole if the contained data items are no longer needed.

Based on this observation, we propose a bloat-aware design paradigm for developing highly efficient Big Data applications. This paradigm includes the following two important components: (1) merging and organizing related small data record objects into few large objects (e.g., byte buffers) instead of representing them explicitly as one-object-per-record, and (2) manipulating data by directly accessing buffers (e.g., at the byte chunk level as opposed to the object level). The central goal of this design paradigm is to bound the number of objects in the application, instead of making it grow proportionally with the cardinality of the input data. It is important to note that these guidelines should be considered explicitly at the early design stage of a Big Data processing system, so that the resulting APIs and implementations would comply with the principles. We have built our own Big Data processing framework Hyracks from the scratch by strictly following this design paradigm. We will use Hyracks a running example to illustrate these design principles.

2.3.1 Data Storage Design: Merging Small Objects

As described in Section 2.2, storing data in Java objects adds much overhead in terms of both memory consumption as well as CPU cycles. As such, we propose to store a group of data items together in *Java memory pages*. Unlike a system-level memory page, which deals with virtual memory, a Java memory page is a fixed-length contiguous block of memory in the (managed) Java heap. For simplicity of presentation, we will use “page” to refer to “Java memory page” in the rest of the chapter. In Hyracks, each page is represented by an object of type `java.nio.ByteBuffer`. Arranging records into pages can reduce the number of objects created in the system from the total number of data items to the number of pages. Hence, the packing factor in such a system can be much closer to that in the ideal representation where data are explicitly laid out in memory and no bookkeeping information needs to be stored. Note that grouping data items into a binary page is just one of many ways to merge

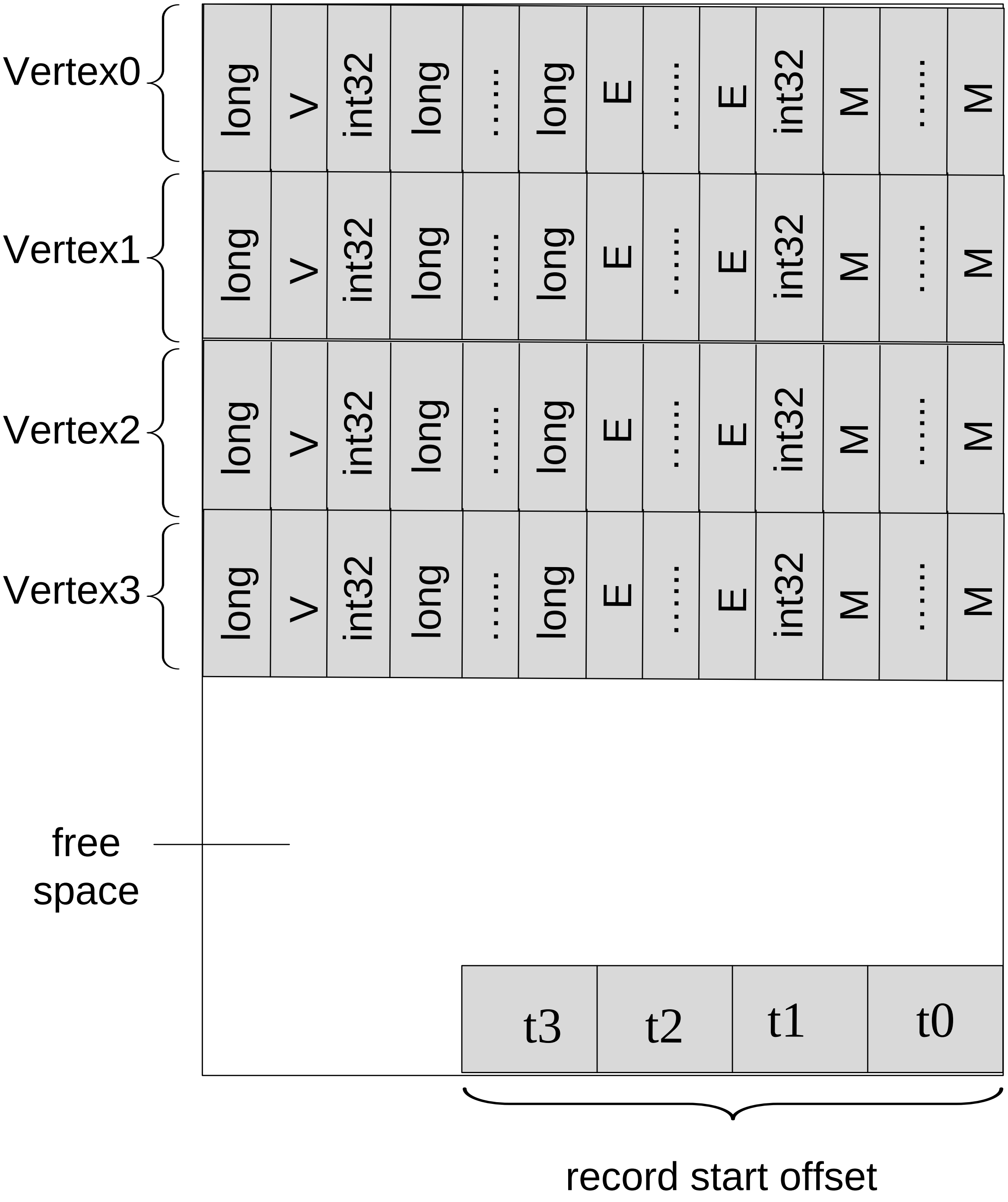


Figure 2.3: Vertices aligning in a page (slots at the end of the page are to support variable-sized vertices).

small objects; other merging (inlining) approaches may also be considered in the future to achieve the same goal.

Multiple ways exist to put records into a page. The Hyracks system takes an approach called "slot-based record management" [84] used widely in the existing DBMS implementations. To illustrate, consider again the PageRank algorithm. Figure 2.3 shows how 4 vertices are stored in a page. It is easy to see that each vertex is stored in its compact layout (as in Figure 2.2) and we use 4 slots (each takes 4 bytes) at the end of the page to store the start offset of each vertex. These offsets will be used to quickly locate data items and support *variable-length* records. Note that the format of data records is invisible to developers, so that they could still focus on high-level data management tasks. Because pages are of fixed size, there is often a small *residual space* that is wasted and cannot be used to store any vertex. To understand the packing factor for this design, we assume each page holds p records on average, and the residual space has r bytes. The overhead for this representation of a vertex includes three parts: the offset slot (4 bytes), the amortized residual space (i.e., r/p), and the amortized overhead of the page object itself (i.e., `java.nio.ByteBuffer`). The page object has an 8-byte header space (in the Oracle 64-bit HotSpot JVM) and a reference (8 bytes) to an internal byte array whose header takes 12 bytes. This makes the amortized overhead of the page object $28/p$. Combining this overhead with the result from Section 2.2.1, we need a total of $(8m + 8n) + 24 + 4 + \frac{r+28}{p}$ bytes to represent a vertex, where $(8m + 8n) + 24$ bytes are used to store the necessary data and $4 + \frac{r+28}{p}$ is the overhead. Because r is the size of the residual space, we have:

$$r \leq 8m + 8n + 24$$

and thus the space overhead of a vertex is bounded by $4 + \frac{8m+8n+52}{p}$. In Hyracks, we use 32KB (as recommended by literature [63]) as the size of a page, and p ranges from 100 to 200 (as seen in experiments with real-world data). To calculate the largest possible overhead, consider the worst case where the residual space has the same size as a vertex. The size of a vertex is thus between $(32768 - 200 * 4)/(200 + 1)$=159 bytes and $(32768 - 100 * 4)/(100 + 1)$=320 bytes. Because the residual space is as large as a vertex, we have $159 \leq r \leq 320$. This leaves the space overhead of a vertex in the range between 4 bytes (because at least 4 bytes are required for the offset slot) and $(4 + (320 + 28)/100)$=7 bytes. Hence, the overall overhead is only $2 - 4\%$ of the actual data size, much less than the 200% overhead of object-based representation (as described in Section 2.2.1).

2.3.2 Data Processor Design: Access Buffers

To support programming with the proposed buffer-based memory management, we propose an *accessor-based* programming pattern. Instead of creating a heap data structure to contain data items and represent their logical relationships, we propose to define an accessor structure that consists of multiple accessors, each to access a different type of data. As such, we need only a very number of accessor structures to process all data, leading to significantly reduced heap objects. In this subsection, we discuss a transformation that can transform regular data structure classes into their corresponding accessor classes. We will also describe the execution model using a few examples.

2.3.2.1 Design Transformations

For each data item class D in a regular object-oriented design, we transform D into an accessor class D_a. Whether a class is a data item class can be specified by the developer. The steps that this transformation includes are as follows.

- *S1*: for each data item field f of type F in D, we add a field f_a of type F_a into D_a, where F_a is the accessor class of F. For each non-data-item field *f'* in D, we copy it directly into D_a.
- *S2*: add a public method `set(byte[] data, int start, int length)` into D_a. The goal of this method is to bind the accessor to a specific byte region where the data items of type D are located. The method can be implemented either by doing *eager materialization*, which recursively binds the binary regions for all its member accessors, or by doing *lazy materialization*, which defers such bindings to the point where the member accessors are actually needed.
- *S3*: for each method M in D, we first create a method M_a which duplicates M in D_a. Next, M_a's signature is modified in a way so that all data-item-type parameters are changed to use their corresponding data accessor types. A parameter accessor provides a way for accessing and manipulating the bound data items in the provided byte regions.

Note that transforming a regular object-oriented design into the above design should be done at the early development stage of a Big Data application in order to avoid the potential development overhead of re-designing after the implementation. Future work will develop compiler support that can automatically transform designs to make them compatible with our memory system.

2.3.2.2 Execution Model

At run time, we form a set of *accessor graphs*, and each accessor graph processes a batch of top-level records. Each node in the graph is an accessor object for a field and each edge represents a "member field" relationship. An accessor graph has the same skeleton as its corresponding heap data structure but does not store any data internally. We let pages flow through the accessor graphs, where a accessor binds to and processes a data item record one-at-a-time. For each thread in the program, the number of accessor graphs needed is equal to the number of data structure types in the program, which is statically bounded. Different instances of a data structure can be processed by the same accessor graph.

If one uses eager materialization in the accessor implementation, the number of accessor objects that need to be created during a data processing task is equal to the total number of nodes in all the accessor graphs. If lazy materialization is chosen to implement accessors, the number of created accessor objects can be significantly reduced, because a member accessor can often be reused for accessing several different data times of the same type. In some cases, additional accessor objects are needed for methods that operate on multiple data items of the same type. For example, a `compare` method defined in a data item class compares two argument data items, and hence, in order to the transformed version of the method would need two accessor objects at run time to perform the comparison. Despite these different ways of implementing accessors, the number of accessor objects needed is always bounded at compile time and does not grow proportionally with the cardinality the dataset.

2.3.2.3 A Running Example

Following the three steps, we manually transform the vertex example in Section 2.2.1 into the following form.

```
public abstract class EdgeListVertexAccessor<
       /** by S1: */
       I extends WritableComparableAccessor,
       V extends WritableAccessor,
       E extends WritableAccessor,
       M extends WritableAccessor>
        extends MutableVertexAccessor<I, V, E, M> {
```

```
private I vertexId = null;

private V vertexValue = null;

/** by S1: indices of its outgoing edges */
private ListAccessor<I> destEdgeIndexList = new
                           ArrayListAccessor<I>();

/** by S1: values of its outgoing edges */
private ListAccessor<E> destEdgeValueList = new
                            ArrayListAccessor<E>();

/** by S1: incoming messages from
    the previous iteration */
private ListAccessor<M> msgList = new
                             ArrayListAccessor<M>();
......

/** by S2:
  * binds the accessor to a binary region
  * of a vertex
  */
public void set(byte[] data, int start, int length){
    /* This may in turn call the set method
       of its member objects. */
       ......
}

/** by S3:  replacing the return type*/
```

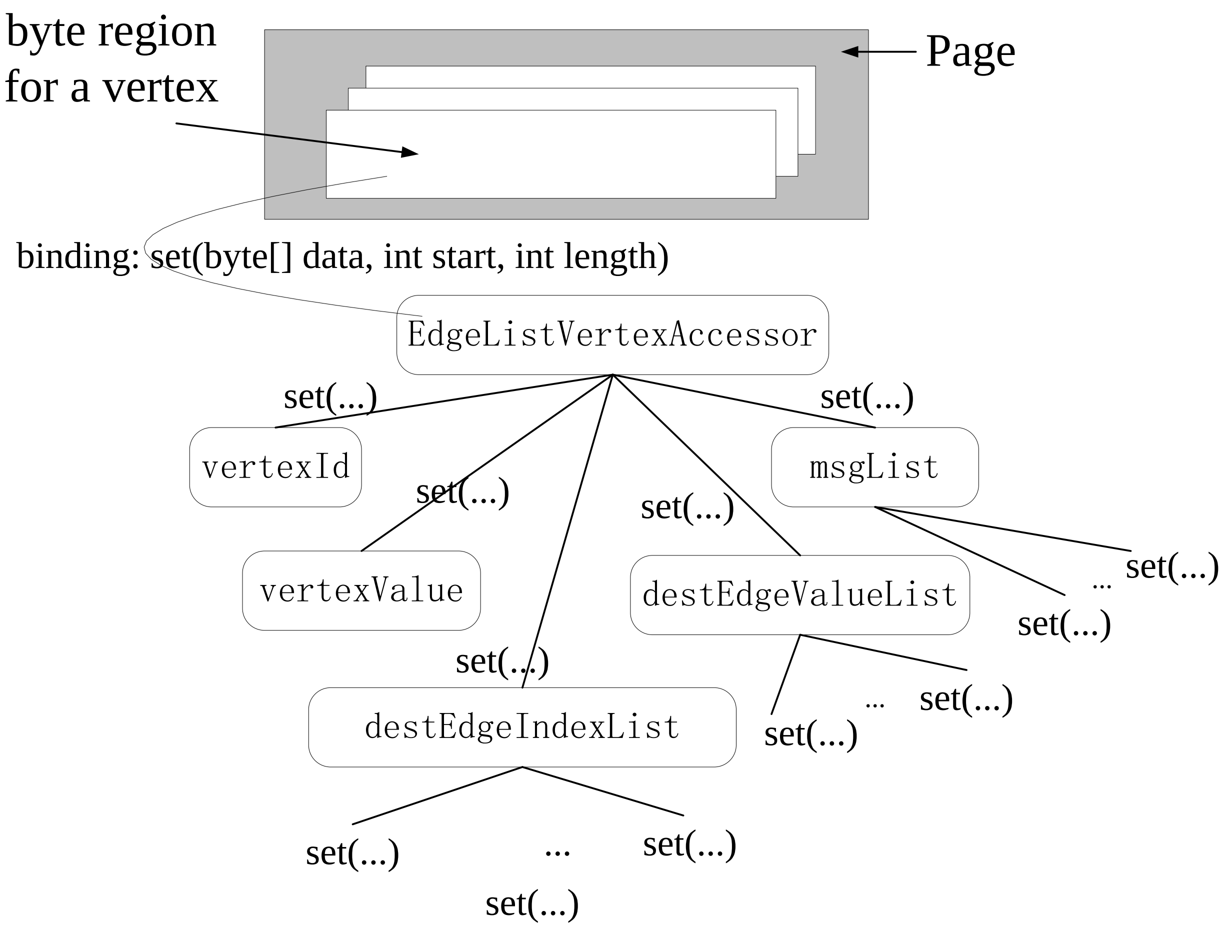


Figure 2.4: A heap snapshot of the example.

```
    public ListAccessor<I> getEdegeIndexes(){
        ...
    }
}
```

In the above code snippet, we highlight the modified code and add the transformation steps as the comments. Figure 2.4 shows a heap snapshot of the running example. The actual data are laid out in pages while an accessor graph is used to process all the vertices, each-at-a-time. For each vertex, the `set` method binds the accessor graph to its byte region.

2.4 Programming Experience

The bloat-aware design paradigm separates the *logical* data access and the *physical* data storage to achieve both compact in-memory data layout and efficient memory management. However, it appears that programming with binary data is a daunting task that creates much burden for the developers. To help reduce the burden, we have developed a comprehensive data management library in Hyracks. In this section, we describe case studies and report our own experience with three real-world projects to show the development effort under this design.

2.4.1 Case Study 1: In-Memory Sort

In this case study, we compare the major parts of two implementations of in-memory sort algorithms, one in Hyracks manipulating binary data using our library, and the other manipulating objects using the standard Java library (e.g., `Collections.sort()`). The code shown below is the method `siftDown`, which is the core component of JDK's sort algorithm.

```
private static void siftDown(Accessor acc, int start, int end) {
   for (int parent = start ; parent < end ; ) {
     int child1 = start + (parent - start) * 2 + 1;
     int child2 = child1 + 1;
     int child = (child2 > end) ? child1
                    : (acc.compare(child1, child2)
                < 0) ? child2 : child1;
     if (child > end)
        break;
     if (acc.compare(parent, child) < 0)
        acc.swap(parent, child);
     parent = child;
  }
}
```

A similar implementation using our library is a segment in the `sort` method, shown as follows:

```
private void sort(int[] tPointers, int offset, int length){
  ......
  while(true){
  ......
     while (c >= b) {
        int cmp = compare(tPointers, c, mi, mj, mv);
        if (cmp < 0) {
           break;
        }
        if (cmp == 0) {
          swap(tPointers, c, d--);
        }
        --c;
     }
     if (b > c)
        break;
     swap(tPointers, b++, c--);
  }
  ......
}
```

The compare method eventually calls a user-defined comparator implementation in which two accessors are bound to the two input byte regions once-at-a-time for the actual comparison. The following code snippet is an example comparator implementation.

```
public class StudentBinaryComparator implements IBinaryComparator{
  private StudentAccessor acc1 = StudentAccessor.getAvailableAccessor();
  private StudentAccessor acc2 = StudentAccessor.getAvailableAccessor();

  public int compare(byte[] data1, int start1, int len1,
     byte[] data2, int start2, int len2){
     acc1.set(data1, start1, len1);
     acc2.set(data2, start2, len2);
     return acc1.compare(acc2);
  }
}
```

Method getAvailableAccessor can be implemented in different ways, depending on the materialization mode. For example, it can directly return a new StudentAccessor object or use an object pool to cache and reuse old objects. As one can tell, the data access code paths in both implementation are well encapsulated with libraries, which allows application developers to focus on the business logic (when to call compare and swap) rather than writing code to access data. As such, the two different implementations have comparable numbers of lines of code.

2.4.2 Case Study 2: Print Records

The second case study is to use a certain format to print data records from a byte array. A typical Java implementation is as follows.

```
private void printData(byte[] data) {
  Reader input = new BufferedReader(new InputStreamReader(
           new DataInputStream(new ByteArrayInputStream(
           data))));
  String line = null;
  while((line = input.readLine())!= null){
    String[] fields = line.split(‘,’);
    //print the record
    ......
  }
  input.close();
 }
}
```

The above implementation reads `String` objects from the input, splits them into fields, and prints them out. In this implementation, the number of created `String` objects is proportional to the cardinality of records multiplied by the number of fields per record. The following code snippet is our printer implementation using the bloat-aware design paradigm.

```
private void printData(byte[] data) {
  PageAccessor pageAcc = PageAccessor.getAvailableAccessor(data);
  RecordPrintingAccessor recordAcc = RecordPrintingAccessor.getAvailableAccessor();
  while(pageAcc.nextRecord()){
    //print the record in the set call
    recordAcc.set(toBinaryRegion(pageAcc));
  }
}
```

In our implementation, we build one accessor graph for printing, let binary data flow through the accessor graph, and print every record by traversing the accessor graph and calling the `set` method on each accessor. It is easy to see that the number of created objects is bounded by the number of nodes of the accessor graph while the programming effort is not significantly increased.

2.4.3 Big Data Projects using the Bloat-Aware Design

The proposed design paradigm has already been used in six Java-based open source Big Data processing systems, listed as follows:

- Hyracks [42] is a data parallel platform that runs data-intensive jobs on a cluster of shared-nothing machines. It executes jobs in the form of directed acyclic graphs that consist of user-defined data processing operators and data redistribution connectors.
- Algebricks [41] is a generalized, extensible, data model-agnostic, and language-independent algebra/optimization layer which is intended to facilitate the implementation of new high-level languages that process Big Data in parallel.
- AsterixDB [29] is a Big Data management system for semi-structured data, which is built on-top-of Algebricks and Hyracks.
- VXQuery [27] is data-parallel XQuery processing engine on top of Algebricks and Hyracks as well.
- Pregelix (will be described Chapter 3) is Big Giraph analytics platform that supports the bulk-synchronous vertex-oriented programming model [75]. It internally uses Hyracks as the run-time execution engine.
- Hivesterix [14] is a SQL-like layer on top of Algebricks and Hyracks; it reuses the modules of the grammar and first-order run-time functions from HiveQL [92].

Project	Overall #LOC	Control #LOC	Data #LOC	Data #LOC Percentage
Hyracks	125930	71227	54803	43.52%
Algebricks	40116	36033	4083	10.17%
AsterixDB	140013	93071	46942	33.53%
VXQuery	45416	19224	26192	57.67%
Pregelix	18411	11958	6453	35.05%
Hivesterix	18503	13910	4593	33.01%
Overall	**388389**	**245323**	**143066**	**36.84%**

Table 2.2: The line-of-code statistics of Big Data projects which uses the bloat-aware design.

In these six projects, the proposed bloat-aware design paradigm is used in the data processing code paths (e.g., the runtime data processing operators such as join, group-by, and sort, as well as the user-defined runtime functions such as plus, minus, sum, and count) while the control code paths (e.g., the runtime control events, the query language parser, the algebra rewriting/optimization, and the job generation module) still use the traditional object-oriented design. Note that for Big Data applications, usually the control path is well isolated from the data path changes (e.g., data format changes) because they execute on different machines — the control path is on the master (controller) machines while the data path is on the slave machines.

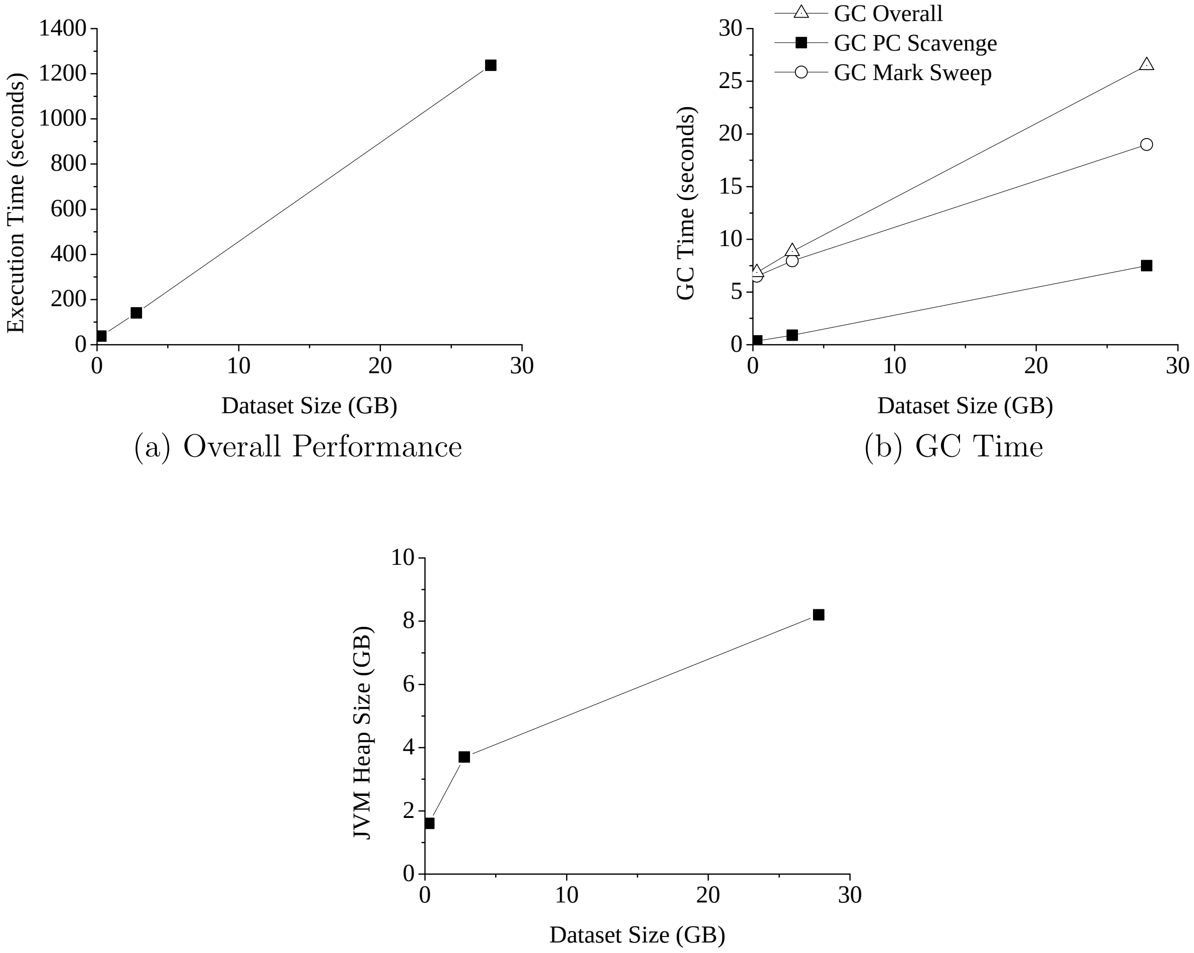
We show a comparison of the lines-of-code (LOC) of the control paths and the data paths in those projects, as listed in Table 2.2. On average, the data path takes about 36.84% of the code base, which is much smaller than the size of the control path. Based on the feedback from the development teams, programming and debugging the data processing components consumes approximately 2× as much time as doing that with the traditional object-orientated design. However, overall, since the data processing components take only 36.84% of the total development effort, the actual development overhead should be much smaller than 2×.

2.4.4 Future Work

The major limitation of the proposed technique is that it requires developers to manipulate low-level, binary representation of data, leading to increased difficulty in programming and debugging. We are in the process of adding another-level-of-indirection to address this issue—we are developing annotation and compiler support to allow for the declarative specifications of data structures. The resulting system will allow developers to annotate data item classes and provide relational specifications. The compiler will automatically generate code that uses the Hyracks library from the user specifications. We hope this system will enable developers to develop high-performance Big Data applications with a low human effort.

2.5 Performance Evaluation

This section presents a set of experiments focusing on performance comparisons between implementations with and without the proposed design paradigm. All experiments were conducted on a cluster of 10 IBM research machines. Each machine has a 4-core Intel Xeon 2.27 GHz processor, 12GB RAM, and 4 300GB 10,000 rpm SATA disk drives, and runs CentOS 5.5 and Java HotSpot(TM) 64-bit server VM (build17.0-b16, mixed node). JVM command line option "-Xmx10g" was used for all our experiments. We use a parallel generational GC in HotSpot, which combines parallel Scavenge (i.e., copying) for the young generation and parallel Mark-Sweep-Compact for the old generation. We collected the application running times and the JVM heap usage statistics in all the 10 machines. Their average is reported in this section. The rest of this section presents experimental studies of our design paradigm on the overall scalability (Section 2.5.1), the packing factors (Section 2.5.2), and the GC costs (Section 2.5.3).



(a) Overall Performance

(b) GC Time

(c) JVM Heap Usage

Figure 2.5: PageRank Performance on Pregelix.

2.5.1 Effectiveness On Overall Scalability: Using PageRank

The Pregelix system (will be presented in Chapter 3) supports nearly the same user interfaces and functionalities as Giraph, but uses the bloat-aware design paradigm. Internally, Pregelix employs a Hyracks runtime dataflow pipeline that contains several data processing operators (i.e., sort, group-by and join) and connectors (i.e., m-to-n hash partitioning merging connector) to execute graph processing jobs. Pregelix supports out-of-core computations, as it uses disk-based operators (e.g., external sort, external group-by, and index outer join) in the dataflow to deal with large amounts of data that cannot be accommodated in the main memory. In Pregelix, both the data storage storage design and data processor design are enabled: it stores all data in pages but employs accessors during data processing. The code shown in Section 2.3.2 is exactly what we used to process vertices.

In order to quantify the potential efficiency gains, we compared the performance of PageRank between Pregelix and Giraph. The goal of this experiment is not to understand the out-of-core performance, but rather to investigate the impact of memory bloat. Therefore, the PageRank algorithm was performed on a subset of Yahoo!'s publicly available AltaVista Web Page Hyperlink Connectivity Graph dataset [105], a snapshot of the World Wide Web from 2002. The subset consists of a total of $561,365,953$ vertices, each of which represents a web page. The size of the decompressed data is 27.8GB, which, if distributed appropriately, should easily fit into the memory of the 10 machines (e.g., 2.78GB for each machine). Each record in the dataset consists of a source vertex identifier and an array of destination vertex identifiers forming the links among different webpages in the graph.

In the experiment, we also ran PageRank with two smaller (1/100 and 1/10 of the original) datasets to obtain the scale-up trend. Figure 2.5 (a), (b), and (c) show, respectively, the overall execution times for the processing of the three datasets, their GC times, and their heap usages. It is easy to see that the total GC time for each dataset is less than 3% of the overall execution time. The JVM heap size is obtained by measuring the overall JVM memory consumption from the operating system. In general, it is slightly larger than the raw data size for each machine (i.e., 2.78GB). This is because extra memory is needed to (1) store object metadata (as we still have objects) and (2) sort and group messages. We also ran the Giraph PageRank implementation on the same three input datasets, but could not succeed for any of the datasets. Giraph crashed with `java.lang.OutOfMemoryError` in all the cases. We confirmed from the Giraph developers that the crash was because of the skewness (e.g., www.yahoo.com has a huge number of inbound links and messages) in the Yahoo! AltaVista Web Page Hyperlink Connectivity Graph dataset, combined with the object-based data representation.

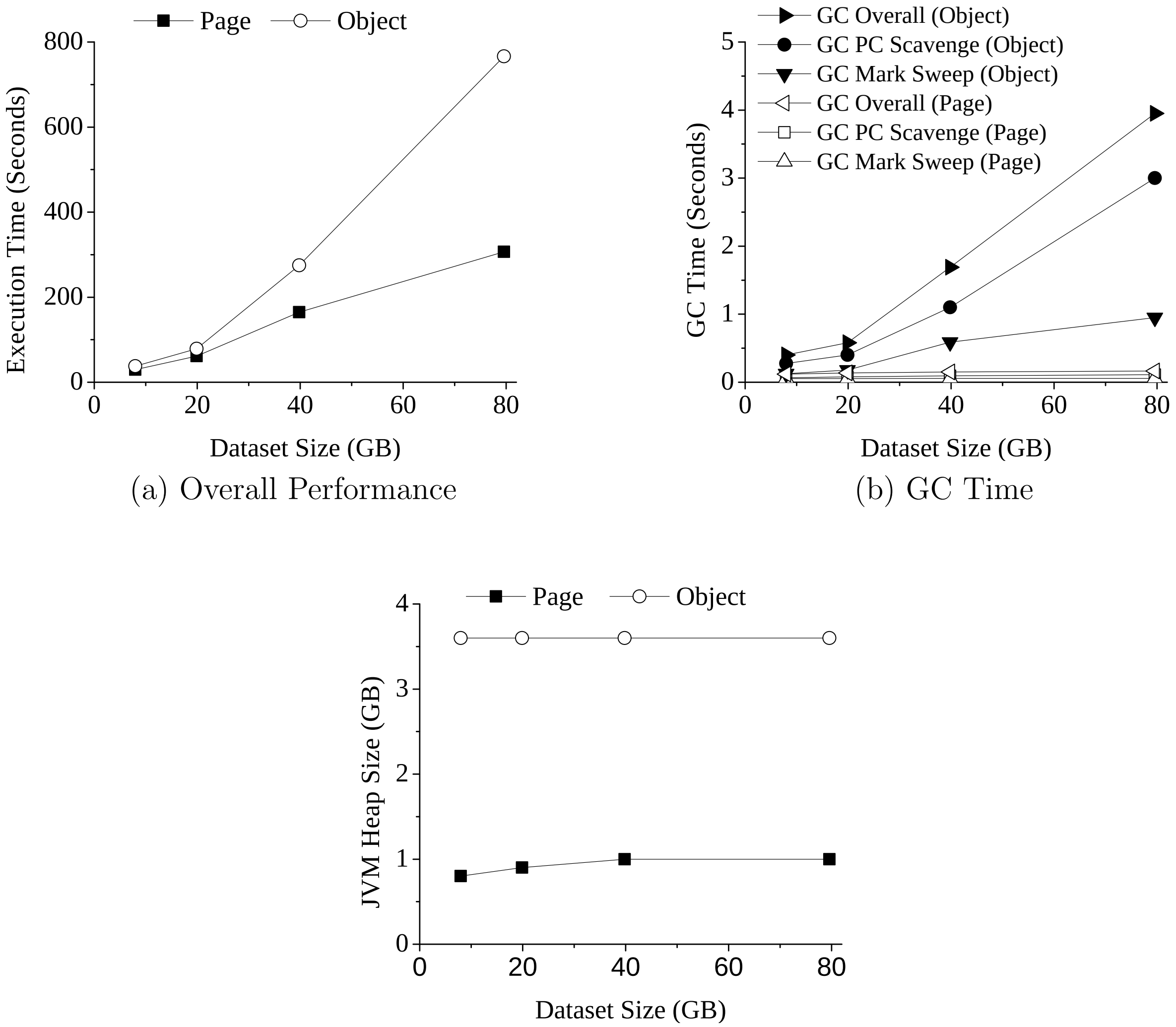
2.5.2 Effectiveness On Packing Factor: Using External Sort

As the second experiment, we compared the performance of two implementations of the standard external sort algorithm [84] on Hyracks. One uses Java objects to represent data records, while the other employs Java memory pages. The goal here is to understand the impact of the low packing factor of the object-based data representation on performance as well as the benefit of the bloat-aware design paradigm.

In the Java object-based implementation, the operator takes deserialized records (i.e., in objects) as input and puts them into a list. Once the number of buffered records exceeds a user-defined threshold, method `sort` is invoked on the list and then the sorted list is dumped to a run file. The processing of the incoming data results in a number of run files. Next, the merge phase starts. This phase reads and merges run files using a priority queue to produce output records. During the merge phase, run files are read into the main memory one-block(32KB)-at-a-time. If the number of files is too large and one pass can not merge all of them, multiple merge passes need be performed.

In the page-based implementation, we never load records from pages into objects. Therefore, the page-based storage removes the header/pointer overhead and improves the packing factor. We implemented a quick sort algorithm to sort in-memory records at the binary level. In this experiment, we used the TPC-H[5] lineitem table as the input data, where each record represents an item in a transaction order. We generated TPC-H data at 10×, 25×, 50× and 100× scales, which correspond to 7.96GB, 19.9GB, 39.8GB and 79.6GB line-item tables, respectively. Each dataset was partitioned among the 10 nodes in a round-robin manner.

[5] The standard data warehousing benchmark, http://www.tpc.org/tpch/



(a) Overall Performance

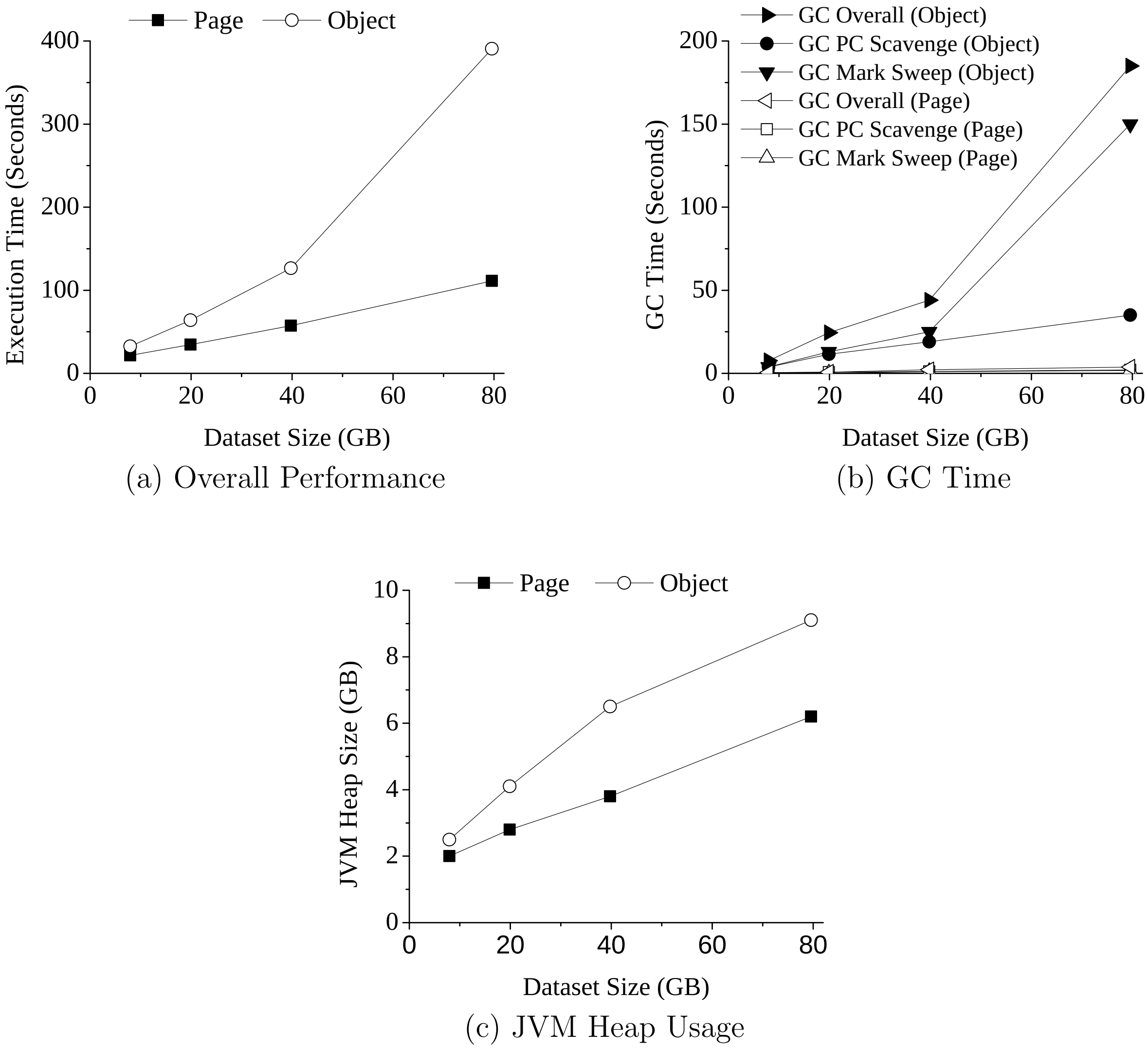
(b) GC Time

(c) JVM Heap Usage

Figure 2.6: ExternalSort Performance.

Particularly, it was divided into 40 partitions, one partition per disk drive (recall that each machine has 4 disk drives).

The task was executed as follows: we created a total of 40 concurrent sorters across the cluster, each of which reads a partition of data locally, sorted them, and wrote the sorted data back to the disk. In this experiment, the page-based implementation used a 32MB sort buffer. The object-based implementation used 5000 the maximal number of in-memory records. The results of two different external sort implementations are plotted in Figure 2.6.



(a) Overall Performance

(b) GC Time

(c) JVM Heap Usage

Figure 2.7: Hash-based Grouping Performance.

The run-time statistics include the overall execution time (Figure 2.6 (a)), the GC overhead (Figure 2.6 (b)), and the JVM heap usage (Figure 2.6 (c)). Note that the Scavenge and Mark-Sweep lines show the GC costs for the young generation and the old generation, respectively. Because sorting a single memory buffer (either 32MB pages or 5000 records) is fast, most data objects are *short-lived and small*, and their lifetimes are usually limited within the iterations where they are created. They rarely get copied into old generations and thus the nursery scans dominate the GC effort.

From Figure 2.6 (a), it is clear to see that as the size of dataset increases, the page-based implementation scales much better (e.g., 2.5× faster on 79.6GB input) than the object-based implementation, and the improvement factor keeps increasing with the increase of the dataset size. The following two factors may contribute to this improvement: (1) due to the low packing factor, the object-based implementation often has more merge passes (hence more I/O) than the page-based implementation, and (2) the use of objects to represent data items leads to increased JVM heap size and hence the operating system has less memory for the file system cache. For example, in the case where the size of the input data is 79.6GB, the page-based implementation has only one merge pass while the object-based one has two merge passes. The page-based implementation also has much less GC time than the object-based implementation (Figure 2.6 (b)). Figure 2.6 (c) shows that the amount of memory required by the object-based implementation is almost 4× larger than that of the page-based implementation, even though the former holds much less actual data in memory (i.e., reflected by the need of an additional merge phase).

2.5.3 Effectiveness On GC Costs: Using Hash-Based Grouping

The goal of the experiment described in this subsection is to understand the GC overhead in the presence of large volumes of objects and references. The input data and scales for this experiment are the same as those in Section 2.5.2. For comparison, we implemented a logical SQL query by hand-coding the physical execution plan and runtime operators, in order to count how many items there are in each order:

```
SELECT l_orderkey, COUNT(*) AS items
FROM lineitem
GROUP BY l_orderkey;
```

We created two different implementations of the underlying hash-based grouping as Hyracks operators. They were exactly the same except that one of them used the object-based hash table (e.g., `java.util.Hashtable`) for grouping and the other used a page-based hash table implementation (e.g., all the intermediate states along with the grouping keys were stored in pages). The hash table size (number of buckets) was 10,485,767 for this experiment, because larger hash tables are often encouraged in Big Data applications to reduce collisions.

The Hyracks job of this task had four operators along the pipeline: a file scan operator (FS), a hash grouping operator (G1), another hash grouping operator (G2), and a file write operator (FW). In the job, FS was locally connected to G1, and then the output of G1 was hash partitioned and fed to G2 (using a m-to-n hash partitioning connector). Finally, G2 was locally connected to FW. Each operator had 40 clones across the cluster, one per partition. Note that G1 was used to group data locally and compress the data in order for it to be sent to the network. G2 performed the final grouping and aggregation.

Figure 2.7 shows the performance of the two different hash-based grouping implementations over the four input datasets. The measurements include the overall performance (Figure 2.7 (a)), the GC time (Figure 2.7 (b)), and the JVM heap size (Figure 2.7(c)). From Figure 2.7(a), one can see that the implementation with the page-based hash table scales much better to the size of data than the other one: in the case where the size of input data is 79.6GB, the former is already 3.5× faster than the latter and the improvement factor keeps growing. In Figure 2.7 (b), we can clearly see that the use of the object-based hash table makes the GC costs go up to 47% of the overall execution time while the page-based hash table does not add much overhead (<3% of the overall execution time). A similar observation can be made on the memory consumption: Figure 2.7 (c) shows that the object-based implementation leads to much larger memory consumption than the page-based one.

Summary Our experimental results clearly demonstrate that, for many data-processing tasks, the bloat-aware design paradigm can lead to far better performance. The source code of all the experiments can be found at: http://code.google.com/p/hyracks.

2.6 Related Work

There exists a large body of work on efficient memory management techniques and data processing algorithms. This section discusses only those that are most closely related to the proposed work. These techniques are classified into three categories: data-processing infrastructures implemented in managed languages, software bloat analysis, and region-based memory management systems.

Java-based data-processing infrastructures Telegraph [88] is a data management system implemented in Java. It uses native byte arrays to store data and has its own memory manager for object allocation, deallocation, and memory de-fragmentation. However, because it is built on top of native memory, developers lose various benefits of a managed language and have to worry about low-level memory correctness issues such as dangling pointers, buffer overflow, and memory leaks. Our approach overcomes the problem by building applications on top of the Java managed memory, providing high efficiency and yet retaining most of the benefits provided by a managed runtime.

Hadoop [10] is a widely-used open source MapReduce implementation. Although it provides a certain level of object reuse for sorting algorithms, its (major) map and reduce interfaces are object-based. The user has to create a great number of objects to implement a new algorithm on top of Hadoop. For example, the reduce-side join implementation in Hive [92] uses a Java `HashMap` in the reduce function to hold the inner branch records, which is very likely to suffer from the same performance problem (i.e., high GC costs) as discussed in Section 2.2.

Other data-processing infrastructures such as Spark [108] and Storm [95] also make heavy use of Java objects in both their core data-processing modules and their application programming interfaces, and thus they are vulnerable to memory bloat as reported in this chapter.

Software bloat analysis Software bloat analysis [80, 79, 102, 100, 77, 89, 103, 99, 31, 78, 101, 104, 98] attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work [80, 79] proposes metrics to provide performance assessment of use of data structures. Their observation that a large portion of the heap is not used to store data is also confirmed in our study. In addition to measure memory usage, our work proposes optimizations specifically targeting the problems we found and our experimental results show that these optimizations are very effective.

Work by Dufour *et al.* [55] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic areas. Shankar *et al.* propose Jolt [89], an approach that makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Work by Xu *et al.* [100] detects memory bloat by profiling copy activities, and their later work [99] looks for high-cost-low-benefit data structures to detect execution bloat. Our work is the first attempt to analyze bloat under the context of Big Data applications and perform effective optimizations to remove bloat.

Region-based memory management Region-based memory management was first used in the implementations of functional languages [94, 28] such as Standard ML [67], and then was extended to Prolog [74], C [65, 68, 60, 61], and real-time Java [39, 72, 43]. More recently, some mark-region hybrid methods such as Immix [40] combine tracing GC with regions to improve GC performance for Java. Our work uses a region-based approach to manage pages, but in a different context — the emerging Big Data applications. Data are stored in pages in the binary form leading to both increased packing factor and decreased memory management cost.

Value types Expanded types in Eiffel and value types in C# are used to declare data with simple structures. However, these types cannot solve the entire bloat problem. In these languages, objects still need to be used to represent data with complicated structures, such as hash maps or lists. The scalability issues that we found in Big Data applications are primarily due to inefficiencies inherent in the object-oriented system design, rather than problems with any specific implementation of a managed language.

2.7 Summary

This chapter presents a bloat-aware design paradigm for Java-based Big Data applications. The study starts with a quantitative analysis of memory bloat using real-world examples, in order for us to understand the impact of excessive object creation on the memory consumption and GC costs. To alleviate this negative influence, we propose a bloat-aware design paradigm, including: merging small objects and accessing data at the binary level. We have performed an extensive set of experiments, and the experimental results have demonstrated that implementations following the design paradigm have much better performance and scalability than the applications that use regular Java objects. The design paradigm can be applied to other managed languages such as C# as well. We believe that the results of this work demonstrate the viability of implementing efficient Big Data applications in a managed, object-oriented language, and open up possibilities for the programming language and systems community to develop novel optimizations targeting data-intensive computing.

This chapter provides a foundation for the next two chapters of this thesis — both the Big Graph analytics system that we will describe in Chapter 3 and the rich(er) graph analytics support in AsterixDB that we will present in Chapter 4 have used the bloat-aware design paradigm in their implementations. The details on where and how the design paradigm is used in those two system works could be found at Section 3.5.4 and Section 4.3.3.

Chapter 3

Pregelix: Big(ger) Graph Analytics on A Dataflow Engine

3.1 Overview

The basic toolkits provided by first-generation “Big Data” Analytics platforms (like Hadoop) lack an essential feature for Big Graph Analytics: MapReduce does not support iteration (or equivalently, recursion) or certain key features required to efficiently iterate “around” a MapReduce program. Moreover, the MapReduce programming model is not ideal for expressing many graph algorithms. This shortcoming has motivated several specialized approaches or libraries that provide support for graph-based iterative programming on large clusters.

Google’s Pregel is a prototypical example of such a platform; it allows problem-solvers to “think like a vertex” by writing a few user-defined functions (UDFs) that operate on vertices, which the framework can then apply to an arbitrarily large graph in a parallel fashion. Open source versions of Pregel have since been developed in the systems community [9][11].

Perhaps unfortunately for both their implementors and users, each such platform is a distinct new system that had to be built from the ground up. Moreover, these systems follow a process-centric design, in which a set of worker processes are assigned partitions (containing sub-graphs) of the graph data and scheduled across a machine cluster. When a worker process launches, it reads its assigned partition into memory, and executes the Pregel (message passing) algorithm. As we will see, such a design can suffer from poor support for problems that are not memory resident. Also desirable, would be the ability to consider alternative runtime strategies that could offer more efficient executions for different sorts of graph algorithms, datasets, and clusters.

The database community has spent nearly three decades building efficient shared-nothing parallel query execution engines [54] that support out-of-core data processing operators (such as join and group-by [64]), and query optimizers [47] that choose an "optimal" execution plan among different alternatives. In addition, deductive database systems—based on Datalog—were proposed to efficiently process recursive queries [37], which can be used to solve graph problems such as transitive closure. However, there is no scalable implementation of Datalog that offers the same fault-tolerant properties supported by today's "Big Data" systems (e.g., [75, 10, 38, 42, 108]). Nevertheless, techniques for evaluating recursive queries—most notably semi-naïve evaluation—still apply and can be used to implement a scalable, fault-tolerant Pregel runtime.

In this chapter, we present Pregelix, a large-scale graph analytics system that we began building in 2011. Pregelix takes a novel set-oriented, iterative dataflow approach to implementing the user-level Pregel programming model. It does so by treating the messages and vertex states in a Pregel computation like tuples with a well-defined schema; it then uses database-style query evaluation techniques to execute the user's program. From a user's perspective, Pregelix provides the same Pregel programming abstraction, just like Giraph [9]. However, from a runtime perspective, Pregelix models Pregel's semantics as a logical query plan and implements those semantics as an iterative dataflow of relational operators that treat message exchange as a join followed by a group-by operation that embeds functions that capture the user's Pregel program. By taking this approach, for the same logical plan, Pregelix is able to offer a set of alternative physical evaluation strategies that can fit various workloads and can be executed by Hyracks [42], a general-purpose shared-nothing dataflow engine (which is also the query execution engine for AsterixDB [29]). By leveraging existing implementations of data-parallel operators and access methods from Hyracks, we have avoided the need to build many critical system components, e.g., bulk-data network transfer, out-of-core operator implementations, buffer managers, index structures, and data shuffle.

To the best of our knowledge, Pregelix is the only Pregel-like system that supports the full Pregel API, runs both in-memory workloads and out-of-core workloads efficiently in a transparent manner on shared-nothing clusters, and provides a rich set of runtime choices. This chapter makes the following contributions:

- An analysis of existing Pregel-like systems: We revisit the Pregel programming abstraction and illustrate some shortcomings of typical custom-constructed Pregel-like systems (Section 3.2).
- A new Pregel architecture: We capture the semantics of Pregel in a logical query plan (Section 3.3), allowing us to execute Pregel as an iterative dataflow.
- A system implementation: We first review the relevant building blocks in Hyracks (Section 3.4). We then present the Pregelix system, elaborating the choices of data storage and physical plans as well as its key implementation details (Section 3.5).
- Case studies: We briefly describe several current use cases of Pregelix from our initial user community (Section 3.6).
- Experimental studies: We experimentally evaluate Pregelix in terms of execution time, scalability, throughput, plan flexibility, and implementation effort (Section 3.7).

3.2 Background and Problems

In this section, we first briefly revisit the Pregel semantics and the Google Pregel runtime (Section 3.2.1) as well as the internals of Giraph, an open source Pregel-like system (Section 3.2.2), and then discuss the shortcomings of such custom-constructed Pregel architectures (Section 3.2.3).

3.2.1 Pregel Semantics and Runtime

Pregel [75] was inspired by Valiant's bulk-synchronous parallel (BSP) model [62]. A Pregel program describes a distributed graph algorithm in terms of vertices, edges, and a sequence of iterations called supersteps. The input to a Pregel computation is a directed graph consisting of edges and vertices; each vertex is associated with a mutable user-defined value and a boolean state indicating its liveness; each edge is associated with a source and destination vertex and a mutable user-defined value. During a superstep S, a user-defined function (UDF) called `compute` is executed at each active vertex V, and can perform any or all of the following actions:

- Read the messages sent to V at the end of superstep $S-1$;
- Generate messages for other vertices, which will be exchanged at the end of superstep S;
- Modify the state of V and its outgoing edges;
- Mutate the graph topology;
- Deactivate V from the execution.

Initially, all vertices are in the active state. A vertex can deactivate itself by "voting to halt" in the call to `compute` using a Pregel provided method. A vertex is reactivated immediately if it receives a message. A Pregel program terminates when every vertex is in the inactive state and no messages are in flight.

In a given superstep, any number of messages may be sent to a given destination. A user-defined `combine` function can be used to pre-aggregate the messages for a destination. In addition, an aggregation function (e.g., min, max, sum, etc.) can be used to compute a global aggregate among a set of participating vertices. Finally, the graph structure can be modified by any vertex; conflicts are handled by using a partial ordering of operations such that all deletions go before insertions, and then by using a user-defined conflict resolution function.

The Google Pregel runtime consists of a centralized master node that coordinates superstep executions on a cluster of worker nodes. At the beginning of a Pregel job, each worker loads an assigned graph partition from a distributed file system. During execution, each worker calls the user-defined `compute` function on each active vertex in its partition, passing in any messages sent to the vertex in the previous superstep; outgoing messages are exchanged among workers. The master is responsible for coordinating supersteps and detecting termination. Fault-tolerance is achieved through checkpointing at user-specified superstep boundaries.

3.2.2 Apache Giraph

Apache Giraph [9] is an open source project that implements the Pregel specification in Java on the Hadoop infrastructure. Giraph launches master and worker instances in a Hadoop map-only job[1], where map tasks run master and worker instances. Once started, the master and worker map tasks internally execute the iterative computation until completion, in a similar manner to Google's Pregel runtime. Figure 3.1 depicts Giraph's process-centric runtime for implementing the Pregel programming model. The vertex data is partitioned across worker tasks (two in this case). Each worker task communicates its control state (e.g., how many active vertices it owns, when it has completed executing a given superstep, etc.)

[1] Alternatively, Giraph can use YARN [96] for resource allocation.

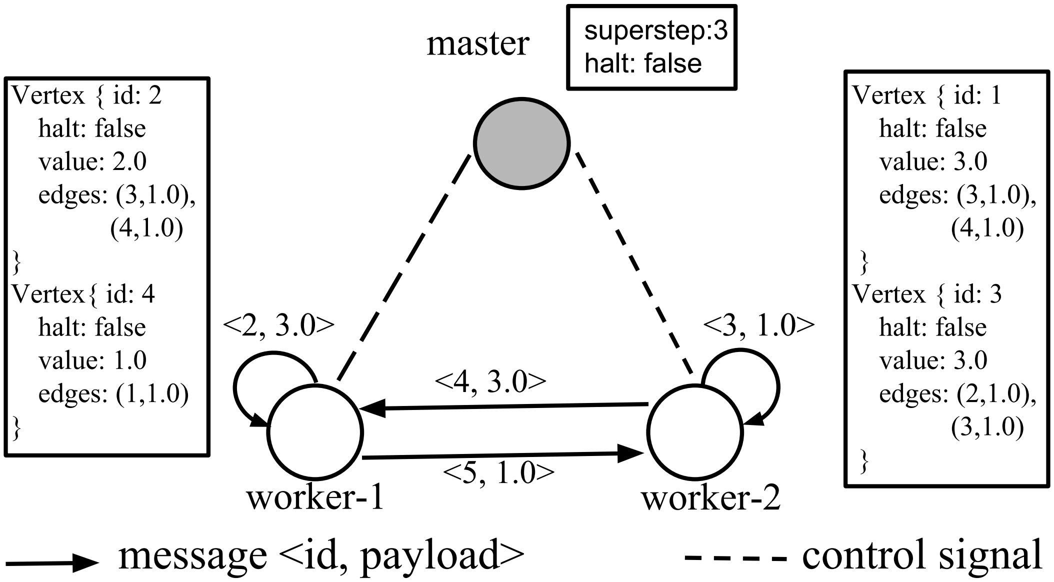


Figure 3.1: Giraph process-centric runtime.

to the master task. The worker tasks establish communication channels between one another for exchanging messages that get sent during individual vertex `compute` calls; some of these messages could be for vertices on the same worker, e.g., messages <2, 3.0> and <3,1.0> in Figure 3.1.

3.2.3 Issues and Opportunities

Most process-centric Pregel-like systems have a minimum requirement for the aggregate RAM needed to run a given algorithm on a particular dataset, making them hard to configure for memory intensive computations and multi-user workloads. In fact, Google's Pregel only supports in-memory computations, as stated in the original paper [75]. Hama [11] has limited support for out-of-core vertex storage using immutable sorted files, but it requires that the messages be memory-resident. The latest version of Giraph has preliminary out-of-core support; however, as we will see in Section 3.7, it does not yet work as expected. Moreover, in the Giraph user mailing list[2] there are 26 cases (among 350 in total) of out-of-memory

[2] http://mail-archives.apache.org/mod_mbox/giraph-user/

related issues from March 2013 to March 2014. The users who posted those questions were typically from academic institutes or small businesses that could not afford memory-rich clusters, but who still wanted to analyze Big Graphs. These issues essentially stem from Giraph's ad-hoc, custom-constructed implementation of disk-based graph processing. This leads to our first opportunity to improve on the current state-of-the-art.

Opportunity (Out-of-core Support) Can we leverage mature database-style storage management and query evaluation techniques to provide better support for out-of-core workloads?

Another aspect of process-centric designs is that they only offer a single physical layer implementation. In those systems, the vertex storage strategy, the message combination algorithm, the message redistribution strategy, and the message delivery mechanism are each usually bound to one specific implementation. Therefore, we cannot choose between alternative implementation strategies that would offer a better fit to a particular dataset, algorithm, cluster or desktop. For instance, the single source shortest paths algorithm exhibits sparsity of messages, in which case a desired runtime strategy could avoid iterating over all vertices by using an extra index to keep track of live vertices. This leads to our second opportunity.

Opportunity (Physical Flexibility) Can we better leverage data, algorithmic, and cluster/hardware properties to optimize a specific Pregel program?

The third issue is that the implementation of a process-centric runtime for the Pregel model spans a full stack of network management, communication protocol, vertex storage, message delivery and combination, memory management, and fault-tolerance; the result is a complex (and hard-to-get-right) runtime system that implements an elegantly simple Pregel semantics. This leads to our third, software engineering opportunity.

Relation	Schema
Vertex	(vid, halt, value, edges)
Msg	(vid, payload)
GS	(halt, aggregate, superstep)

Table 3.1: Nested relational schema that models the Pregel state.

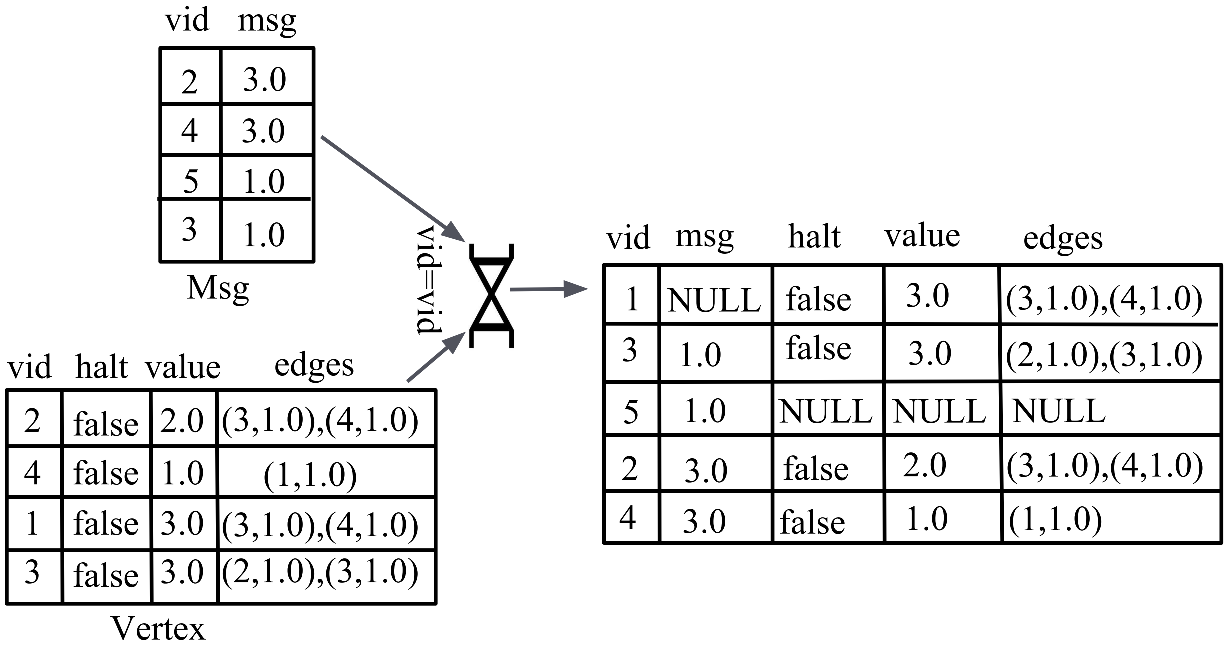
Opportunity (Software Simplicity) Can we leverage more from existing data-parallel platforms—platforms that have been improved for many years—to simplify the implementation of a Pregel-like system?

We will see how these opportunities are exploited by our proposed architecture and implementation in Section 3.5.8.

3.3 The Pregel Logical Plan

In this section, we model the semantics of Pregel as a logical query plan. This model will guide the detailed design of the Pregelix system (Section 3.5).

Our high level approach is to treat messages and vertices as data tuples and use a join operation to model the message passing between vertices, as depicted in Figure 3.2. Table 3.1 defines a set of nested relations that we use to model the state of a Pregel execution. The input data is modeled as an instance of the `Vertex` relation; each row identifies a single vertex with its halt, value, and edge states. All vertices with a $halt = false$ state are active in the current superstep. The value and edges attributes represent the vertex state and neighbor list, which can each be of a user-defined type. The messages exchanged between vertices in a superstep are modeled by an instance of the `Msg` relation, which associates a destination vertex identifier with a message payload. Finally, the `GS` relation from Table 3.1 models



vid	msg
2	3.0
4	3.0
5	1.0
3	1.0

Msg

vid	halt	value	edges
2	false	2.0	(3,1.0),(4,1.0)
4	false	1.0	(1,1.0)
1	false	3.0	(3,1.0),(4,1.0)
3	false	3.0	(2,1.0),(3,1.0)

Vertex

vid	msg	halt	value	edges
1	NULL	false	3.0	(3,1.0),(4,1.0)
3	1.0	false	3.0	(2,1.0),(3,1.0)
5	1.0	NULL	NULL	NULL
2	3.0	false	2.0	(3,1.0),(4,1.0)
4	3.0	false	1.0	(1,1.0)

Figure 3.2: Implementing message-passing as a logical join.

the global state of the Pregel program; here, when *halt* = *true* the program terminates[3], *aggregate* is a global state value, and *superstep* tracks the current iteration count.

Figure 3.2 models message passing as a join between the `Msg` and `Vertex` relations. A full-outer-join is used to match messages with vertices corresponding to the Pregel semantics as follows:

- The inner case matches incoming messages with existing destination vertices;
- The left-outer case captures messages sent to vertices that may not exist; in this case, a vertex with the given *vid* is created with other fields set to NULL.
- The right-outer case captures vertices that have no messages; in this case, `compute` still needs to be called for such a vertex if it is active.

[3] This global halting state depends on the halting states of all vertices as well as the existence of messages.

UDF	Description
compute	Executed at each active vertex in every superstep.
combine	Aggregation function for messages.
aggregate	Aggregation function for the global state.
resolve	Used to resolve conflicts in graph mutations.

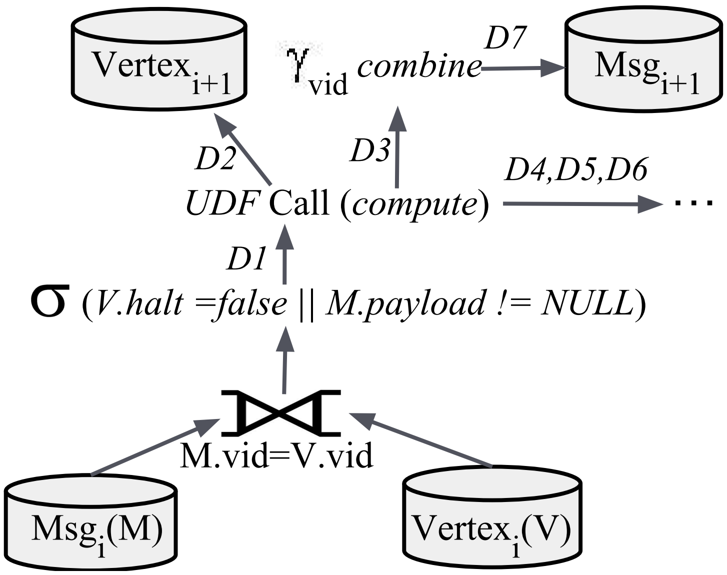
Table 3.2: UDFs used to capture a Pregel program.

The output of the full-outer-join will be sent to further operator processing steps that implement the Pregel semantics; some of these downstream operators will involve UDFs that capture the details (e.g., `compute` implementation) of the given Pregel program.

Table 3.2 lists the UDFs that implement a given Pregel program. In a given superstep, each active vertex is processed through a call to the `compute` UDF, which is passed the messages sent to the vertex in the previous superstep. The output of a call to `compute` is a tuple that contains the following fields:

- The possibly updated `Vertex` tuple.
- A list of outbound messages (delivered in the next superstep).
- The global *halt* state contribution, which is *true* when the outbound message list is empty and the *halt* field of the updated vertex is *true*, and *false* otherwise.
- The global *aggregate* state contribution (tuples nested in bag).
- The graph mutations (a nested bag of tuples to insert/delete to/from the `Vertex` relation).

As we will see below, this output is routed to downstream operators that extract (project) one or more of these fields and execute the dataflow of a superstep. For instance, output messages are grouped by the destination vertex id and aggregated by the `combine` UDF. The global aggregate state contributions of all vertices are passed to the `aggregate` function, which produces the global aggregate state value for the subsequent superstep. Finally, the `resolve` UDF accepts all graph mutations—expressed as insertion/deletion tuples against



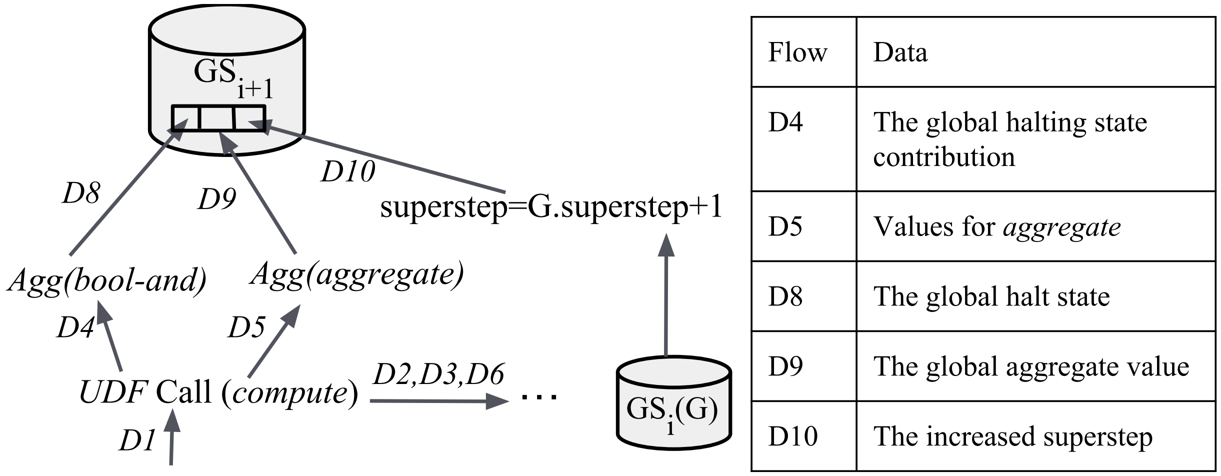
Flow	Data
D2	Vertex tuples
D3	Msg tuples
D7	Msg tuples after combination

Figure 3.3: The basic logical query plan of a Pregel superstep i which reads the data generated from the last superstep (e.g., `Vertex`$_i$, `Msg`$_i$, and `GS`$_i$) and produces the data (e.g., `Vertex`$_{i+1}$, `Msg`$_{i+1}$, and `GS`$_{i+1}$) for superstep $i + 1$. Global aggregation and synchronization are in Figure 3.4, and vertex addition and removal are in Figure 3.5.

the `Vertex` relation—as input, and it resolves any conflicts before they are applied to the `Vertex` relation.

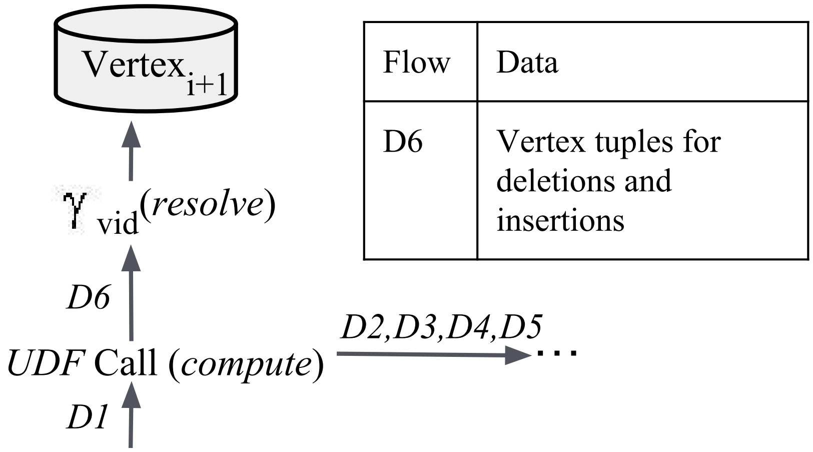
We now turn to the description of a single logical dataflow plan; we divide it into three figures that each focus on a specific application of the (shared) output of the `compute` function. The relevant dataflows are labeled in each figure. Figure 3.3 defines the input to the `compute` UDF, the output messages, and updated vertices. Flow $D1$ describes the `compute` input for superstep i as being the output of a full-outer-join between `Msg` and `Vertex` (as described by Figure 3.2) followed by a selection predicate that prunes inactive vertices. The `compute` output pertaining to vertex data is projected onto dataflow $D2$, which then updates the `Vertex` relation. In datafow $D3$, the message output is grouped by destination vertex id and aggregated by the `combine` function[4], which produces flow $D7$ that is inserted into the `Msg` relation.

[4] The default `combine` gathers all messages for a given destination into a list.



Flow	Data
D4	The global halting state contribution
D5	Values for *aggregate*
D8	The global halt state
D9	The global aggregate value
D10	The increased superstep

Figure 3.4: The plan segment that revises the global state.



Flow	Data
D6	Vertex tuples for deletions and insertions

Figure 3.5: The plan segment for vertex addition/removal.

The global state relation `GS` contains a single tuple whose fields comprise the global state. Figure 3.4 describes the flows that revise these fields in each superstep. The halt state and global aggregate fields depend on the output of `compute`, while the superstep counter is simply its previous value plus one. Flow $D4$ applies a boolean aggregate function (logical AND) to the global halting state contribution from each vertex; the output (flow $D8$) indicates the global halt state, which controls the execution of another superstep. Flow $D5$ routes the global aggregate state contributions from all active vertices to the `aggregate` UDF which then produces the global aggregate value (flow $D9$) for the next superstep.

Graph mutations are specified by a `Vertex` tuple with an operation that indicates insertion (adding a new vertex) or deletion (removing a vertex)[5]. Flow $D6$ in Figure 3.5 groups these mutation tuples by vertex id and applies the `resolve` function to each group. The output is then applied to the `Vertex` relation.

3.4 The Runtime Platform

The Pregel logical plan could be implemented on any parallel dataflow engine, including Stratosphere [38], Spark [108] or Hyracks [42]. As we argue below, we believe that Hyracks is particularly well-suited for this style of computation; this belief is supported by Section 3.7's experimental results (where some of the systems studied are based on other platforms). The rest of this section covers the Hyracks platform [42], which is Pregelix's target runtime for the logical plan in Section 3.3. Hyracks is a data-parallel runtime in the same general space as Hadoop [10] and Dryad [70]. Jobs are submitted to Hyracks in the form of DAGs (directed acyclic graphs) that are made up of operators and connectors. Operators are responsible for consuming input partitions and producing output partitions. Connectors perform redistributions of data between operators. For a submitted job, in a Hyracks cluster, a master machine dictates a set of worker machines to execute clones of the operator DAG in parallel and orchestrates data exchanges. Below, we enumerate the features and components of Hyracks that we leverage to implement the logical plan described in Section 3.3.

User-configurable task scheduling. The Hyracks engine allows users to express task scheduling constraints (e.g., count constraints, location choice constraints, or absolute location constraints) for each physical operator. The task scheduler of Hyracks is a constraint solver that comes up with a schedule satisfying the user-defined constraints. In Section 3.5.3.4, we leverage this feature of Hyracks to implement sticky, iterative dataflows.

[5] Pregelix leaves the application-specific vertex deletion semantics in terms of integrity constraints to application programmers.

Access methods. B-trees and LSM B-trees are part of the Hyracks storage library. A B-tree [50] is a commonly used index structure in most commercial databases; it supports efficient lookup and scan operations, but a single tree update can cause random I/Os. In contrast, the LSM B-tree [82] puts updates into an in-memory component (e.g., an in-memory B-tree); it merges the in-memory component with disk components in a periodic manner, which turns random I/Os for updates into sequential ones. The LSM B-tree thus allows fast updates but may result in slightly slower lookups.

Group-by operators. The Hyracks operator library includes three group-by operator implementations: sort-based group-by, which pushes group-by aggregations into both the in-memory sort phase and the merge phase of an external sort operator; HashSort group-by, which does the same thing as the sort-based one except using hash-based group-by for the in-memory processing phase; and preclustered group-by, which assumes incoming tuples are already clustered by the group-by key and hence just applies the aggregation operation in sequence to one group after the other.

Join operators. Merge-based index full outer join and probe-based index left-outer join are supported in the Hyracks operator library. The full outer join operator merges sorted input from the outer relation with an index scan on the inner relation; tuples containing `NULL` values for missing fields will be generated for no-matches. The left-outer join operator, for each input tuple in the outer relation, consults an index on the inner relation for matches that produce join results or for no-matches that produce tuples with `NULL` values for the inner relation attributes.

Connectors. Hyracks connectors define inter-operator data exchange patterns. Here, we focus on the following three communication patterns: an m-to-n partitioning connector repartitions the data based on a user-defined partitioning function from m (sender-side) partitions to n (receiver-side) partitions; the m-to-n partitioning merging connector does the same thing but assumes tuples from the sender-side are ordered and therefore simply merges the input streams at the receiver-side; the aggregator connector reduces all input streams to a single receiver partition.

Materialization policies. We use two materialization policies that Hyracks supports for customizing connectors: fully pipelined, where the data from a producer is immediately pushed to the consumer, and sender-side materializing pipelined, where the data transfer channel launches two threads at the sender side, one that writes output data to a local temporary file, and another that pulls written data from the file and sends it to the receiver-side.

3.5 The Pregelix System

In this section, we describe our implementation of the logical plan (Section 3.3) on the Hyracks runtime (Section 3.4), which is core of the Pregelix system. We elaborate on data-parallel execution (Section 3.5.1), data storage (Section 3.5.2), physical query plan alternatives (Section 3.5.3), memory management (Section 3.5.4), fault-tolerance (Section 3.5.5), and job pipelining (Section 3.5.6). We conclude by summarizing the software components of Pregelix (Section 3.5.7) and revisiting our three opportunities (Section 3.5.8).

3.5.1 Parallelism

To parallelize the logical plan of the Pregel computation described in Section 3.3 at runtime, one or more clones of a physical plan—that implements the logical plan—are shipped to Hyracks worker machines that run in parallel. Each clone deals with a single data partition. During execution, data is exchanged from the clones of an upstream operator to those of a downstream operator through a Hyracks connector. Figure 3.6 shows an example, where the logical join described in Figure 3.2 is parallelized onto two workers and message tuples

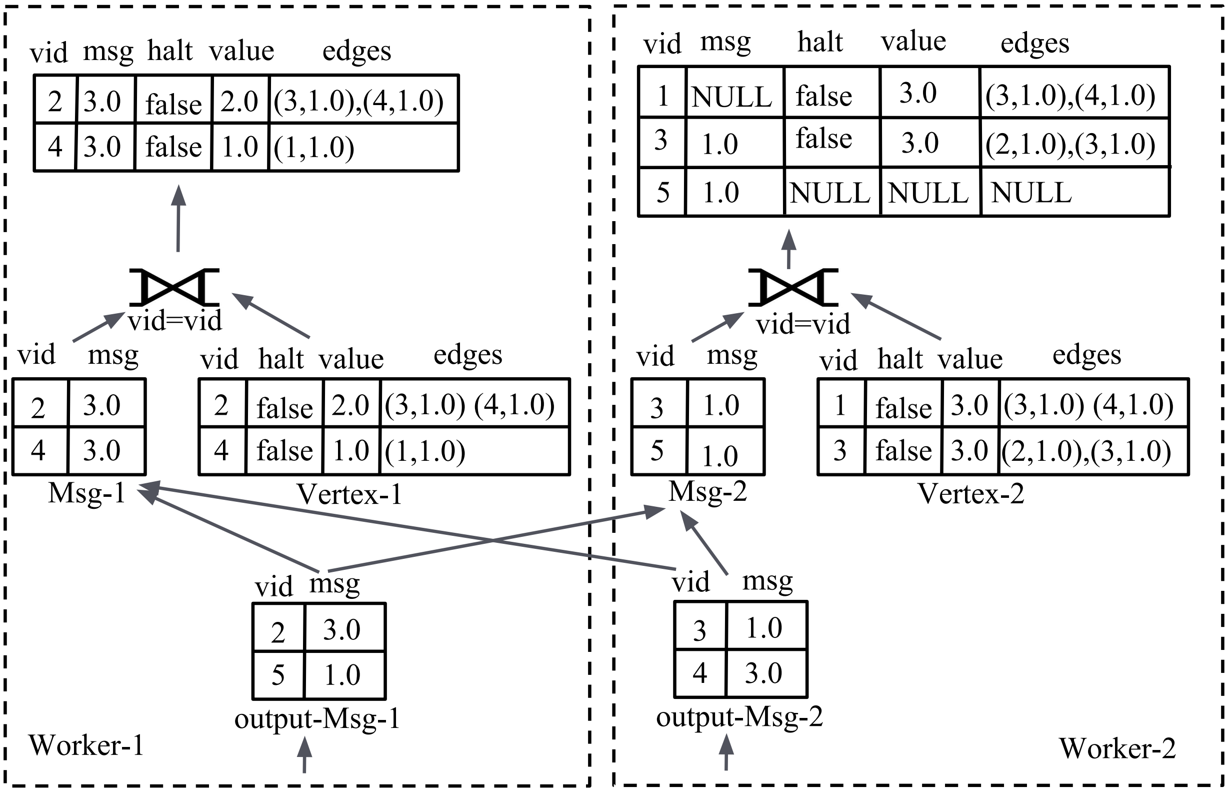


Figure 3.6: The parallelized join for the logical join in Figure 3.2.

are exchanged from producer partitions (operator clones) to consumer partitions (operator clones) using an m-to-n partitioning connector, where m and n are equal to two.

3.5.2 Data Storage

Given a graph analytical job, Pregelix first loads the input graph dataset (the initial `Vertex` relation) from a distributed file system, i.e., HDFS, into a Hyracks cluster, partitioning it by *vid* using a user-defined partitioning function[6] across the worker machines. After the eventual completion of the overall Pregel computation, the partitioned `Vertex` relation is scanned and dumped back to HDFS. During the supersteps, at each worker node, one (or more) local indexes—keyed off of the *vid* field—are used to store one (or more) partitions of the `Vertex` relation. Pregelix leverages both B-tree and LSM B-tree index structures

[6] By default, we use hash partitioning.

from the Hyracks storage library to store partitions of `Vertex` on worker machines. The choice of which index structure to use is workload-dependent and user-selectable. A B-tree index performs well on jobs that frequently update vertex data in-place, e.g., PageRank. An LSM B-tree index performs well when the size of vertex data is changed drastically from superstep to superstep, or when the algorithm performs frequent graph mutations, e.g., the path merging algorithm in genome assemblers [109].

The `Msg` relation is initially empty; it is refreshed at the end of a superstep with the result of the message `combine` function call in the (logical) dataflow $D7$ of Figure 3.3; the physical plan is described in Section 3.5.3.1. The message data is partitioned by destination vertex id (*vid*) using the same partitioning function applied to the vertex data, and is thus stored (in temporary local files) on worker nodes that maintain the destination vertex data. Furthermore, each message partition is sorted by the *vid* field.

Lastly, we leverage HDFS to store the global state of a Pregelix job; an access method is used to read and cache the global state at worker nodes when it is referenced by user-defined functions like `compute`.

3.5.3 Physical Query Plans

In this subsection, we dive into the details of the physical plans for the logical plan described in Figures 3.3, 3.4, and 3.5. Our discussion will cover message combination and delivery, global states, graph mutations, and data redistribution.

3.5.3.1 Message Combination

Figure 3.3 uses a logical group-by operator for message combination. For that, Pregelix leverages the three group-by operator implementations mentioned in Section 3.4. A preclustered

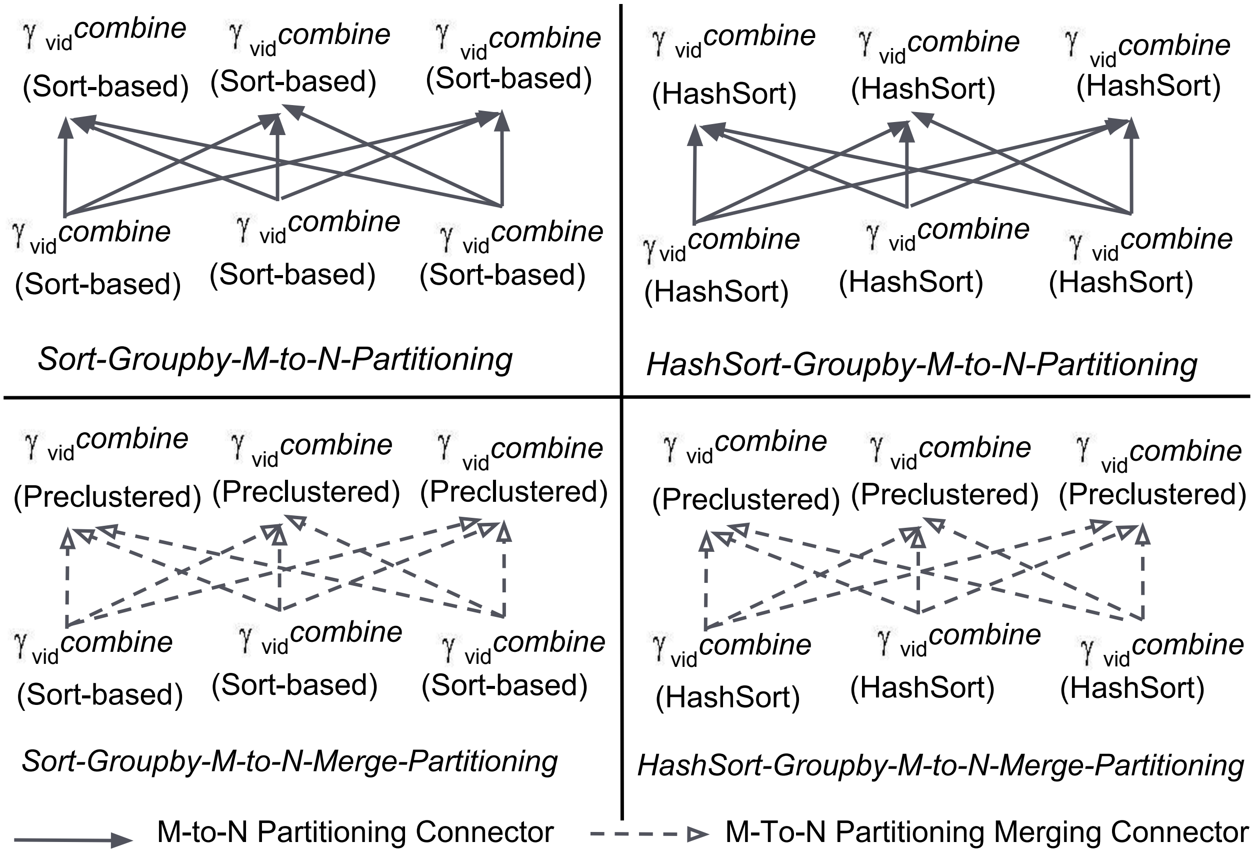

γ vid combine
(Sort-based)
Sort-Groupby-M-to-N-Partitioning
γ vid combine
(HashSort)
HashSort-Groupby-M-to-N-Partitioning
γ vid combine
(Preclustered)
Sort-Groupby-M-to-N-Merge-Partitioning
HashSort-Groupby-M-to-N-Merge-Partitioning
M-to-N Partitioning Connector
M-To-N Partitioning Merging Connector

Figure 3.7: The four physical group-by strategies for the group-by operator which combines messages in Figure 3.3.

group-by can only be applied to input data that is already clustered by the grouping key. A HashSort group-by operator offers better performance (over sort-based group-by) when the number of groups (the number of distinct message receivers in our case) is small; otherwise, these two group-by operators perform similarly. In a parallel execution, the grouping is done by two stages—each producer partitions its output (message) data by destination *vid*, and the output is redistributed (according to destination *vid*) to each consumer, which performs the final grouping step.

Pregelix has four different parallel group-by strategies, as shown in Figure 3.7. The lower two strategies use an m-to-n partitioning merging connector and only need a simple one-pass pre-clustered group-by at the receiver-side; however, in this case, receiver-side merging needs to coordinate the input streams, which takes more time as the cluster size grows.

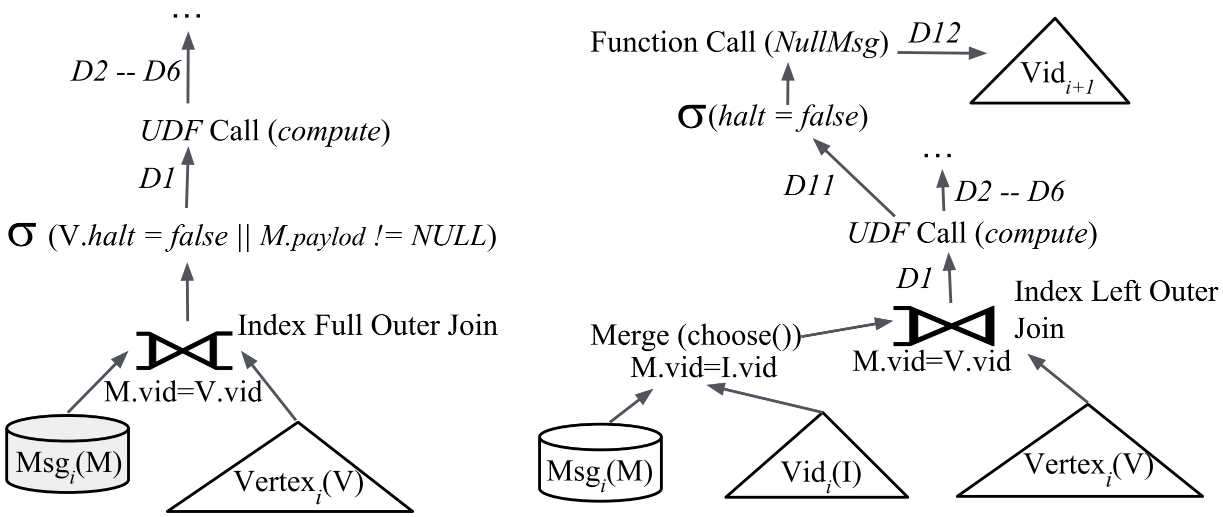


Figure 3.8: Two physical join strategies for forming the input to the `compute` UDF. On the left is an index full outer join approach. On the right is an index left outer join approach.

The upper two strategies use an m-to-n partitioning connector, which does not require such coordination; however, these strategies do not deliver sorted data streams to the receiver-side group-bys, so re-grouping is needed at the receiver-side. A fully pipelined policy is used for the m-to-n partitioning connector in the upper two strategies, while in the lower two strategies, a sender-side materializing pipelined policy is used by the m-to-n partitioning merging connector to avoid possible deadlock scenarios mentioned in the query scheduling literature [64]. The choice of which group-by strategy to use depends on the dataset, graph algorithm, and cluster. We will further pursue this choice in our experiments (Section 3.7).

3.5.3.2 Message Delivery

Recall that in Figure 3.3, a logical full-outer-join is used to deliver messages to the right vertices and form the data needed to call the `compute` UDF. For that, we use index-based joins because (a) vertices are already indexed by vid, and (b) all four group-by strategies in Figure 3.7 flow the (combined) messages out of their receiver-side group-bys in vid-sorted order, thereby producing vid-sorted Msg partitions.

Pregelix offers two physical choices for index-based joins—an index full outer join approach and an index left outer join approach, as shown in Figure 3.8. The full outer join plan scans the entire vertex index to merge it with the (combined) messages. This join strategy is suitable for algorithms where most vertices are live (active) across supersteps (e.g., PageRank). The left outer join plan prunes unnecessary vertex scans by first searching the live vertex index for each (combined) incoming message, and it fits cases where messages are sparse and only few vertices are live in every superstep (e.g., single source shortest paths). A user can control which join approach Pregelix uses for a given job. We now briefly explain the details of the two join approaches.

Index Full Outer Join. As shown in left side of Figure 3.8, this plan is straightforwardly mapped from the logical plan. The join operator simply merges a partition of `Msg` and `Vertex` using a single pass.

Index Left Outer Join. As shown in right of Figure 3.8, this plan initially bulk loads another B-tree `Vid` with null messages (vid, NULL) that are generated by a function `NullMsg`. This index serves to represent the set of currently active vertices. The dataflows $D11$ and $D12$ in Figure 3.8 are (vid, halt) tuples and (vid, NULL) tuples respectively. Note that `Vid` is partitioned in the same way as `Vertex`. In the next superstep, a merge operator merges tuples from `Msg` and `Vid` based on the equivalence of the `vid` fields, and the *choose* function inside the operator selects tuples from `Msg` to output when there are duplicates. Output tuples of the merge operator are sent to an index left outer join operator that probes the `Vertex` index. Tuples produced by the left outer join operator are directly sent to the `compute` UDF. The original filter operator $\sigma(\mathit{V.halt{=}false}||\mathit{M.payload!{=}\ NULL})$ in the logical plan is logically transformed to the merge operator where tuples in `Vid` satisfy *halt=false* and tuples in `Msg` satisfy *M.payload!= NULL*.

To minimize data movements among operators, in a physical plan, we push the filter operator, the UDF call of `compute`, the update to `Vertex`, and the extraction (project) of fields in the output tuple of `compute` into the upstream join operator as Hyracks "mini-operators."

3.5.3.3 Global States and Graph Mutations

To form the global halt state and aggregate state—see the two global aggregations in Figure 3.4—we leverage a standard two-stage aggregation strategy. Each worker pre-aggregates these state values (stage one) and sends the result to a global aggregator that produces the final result and writes it to HDFS (stage two). The incrementing of superstep is also done by a trivial dataflow.

The additions and removals of vertices in Figure 3.5 are applied to the `Vertex` relation by an index insert-delete operator. For the group-by operator in Figure 3.5, we only do a receiver-side group-by because the `resolve` function is not guaranteed to be distributive and the connector for $D6$ (in Figure 3.3) is an m-to-n partitioning connector in the physical plan.

3.5.3.4 Data Redistribution

In a physical query plan, data redistribution is achieved by either the m-to-n hash partitioning connector or the m-to-n hash partitioning merging connector (mentioned in Section 3.4). With the Hyracks provided user-configurable scheduling, we let the location constraints of the join operator (in Figure 3.8) be the same as the places where partitions of `Vertex` are stored across all the supersteps. Also, the group-by operator (in Figure 3.7) has the same location constraints as the join operator, such that in all supersteps, `Msg` and `Vertex` are partitioned in the same (sticky) way and the join between them can be done without extra repartitioning. Therefore, the only necessary data redistributions in a superstep are (a) redistributing outgoing (combined) messages from sender partitions to the right vertex partitions, and (b) sending each vertex mutation to the right partition for addition or removal in the graph data.

3.5.4 Memory Management

Hyracks operators and access methods already provide support for out-of-core computations. The default Hyracks memory parameters work for all aggregated memory sizes as long as there is sufficient disk space on the worker machines. To support both in-memory and out-of-core workloads, B-trees and LSM-trees both leverage a buffer cache that caches partition pages and gracefully spills to disk only when necessary using a standard replacement policy, i.e., LRU. In the case of an LSM B-tree, some number of buffer pages are pinned in memory to hold memory-resident B-tree components.

The majority of the physical memory on a worker machine is divided into four parts: the buffer cache for access methods of the `Vertex` relation; the buffers for the group-by operator clones; the buffers for network channels; and the file system cache for (a) temporary run files generated by group-by operator clones, (b) temporary files for materialized data redistributions, and (c) temporary files for the relation `Msg`. The first three memory components are explicitly controlled by Pregelix and can be tuned by a user, while the last component is (implicitly) managed by the underlying OS. Although the Hyracks runtime is written in Java, it uses the bloat-aware design that we proposed in Chapter 2 to avoid unnecessary memory bloat and to minimize the performance impact of garbage collection in the JVM.

3.5.5 Fault-Tolerance

Pregelix offers the same level of fault-tolerance as other Pregel-like systems [75, 9, 11] by checkpointing states to HDFS at user-selected superstep boundaries. In our case, the states to be checkpointed at the end of a superstep include Vertex and Msg (as well as Vid if the left outer join approach is used). The checkpointing of Msg ensures that a user program does not need to be aware of failures. Since GS stores its primary copy in HDFS, it need not be part of the checkpoint. A user job can determine whether or not to checkpoint after a superstep. Once a node failure or disk failure happens, the failed machine is added into a blacklist.

During recovery, Pregelix finds the latest checkpoint and reloads the states to a newly selected set of failure-free worker machines. Reloading states includes two steps. First, it kicks off physical query plans to scan, partition, sort, and bulk load the entire Vertex and Vid (if any) from the checkpoint into B-trees (or LSM B-trees), one per partition. Second, it executes another physical query plan to scan, partition, sort, and write the checkpointed Msg data to each partition as a local file.

3.5.6 Job Pipelining

Pregelix can accept an array of jobs and pipeline between compatible contiguous jobs without HDFS writes/reads nor index bulk-loads. Two compatible jobs should have a producer-consumer relationship regarding the output/input data and share the same type of vertex—meaning, they interpret the corresponding bits in the same way. This feature was motivated by the genome assembler [109] application which runs six different graph cleaning algorithms that are chained together for many rounds. A user can choose to enable this option to get improved performance with reduced fault-tolerance.

3.5.7 Pregelix Software Components

Pregelix supports the Pregel API introduced in Section 3.2.1 in Java, which is very similar to the APIs of Giraph [9] and Hama [11]. Internally, Pregelix has a statistics collector, failure manager, scheduler, and plan generator which run on a client machine after a job is submitted; it also has a runtime context that stays on each worker machine. We describe each component below.

Statistics Collector. The statistics collector periodically collects statistics from the target Hyracks cluster, including system-wide counters such as CPU load, memory consumption, I/O rate, network usage of each worker machine, and the live machine set, as well as Pregel-specific statistics such as the vertex count, live vertex count, edge count, and message count of a submitted job.

Failure Manager. The failure manager analyzes failure traces and recovers from those failures that are indeed recoverable. It only tries to recover from interruption errors (e.g., a worker machine is powered off) and I/O related failures (e.g., disk I/O errors); it just forwards application exceptions to end users. Recovery is done as mentioned in Section 3.5.5.

Scheduler. Based on the information obtained by the statistics collector and the failure manager, the scheduler determines which worker machines to use to run a given job. The scheduler assigns as many partitions to a selected machine as the number of its cores. For each Pregel superstep, Pregelix sets location constraints for operators in the manner mentioned in Section 3.5.3.4. For loading `Vertex` from HDFS [10], the constraints of the data scanning operator (set by the scheduler) exploit data locality for efficiency.

Plan Generator. The plan generator generates physical query plans for data loading, result writing, each single Pregel superstep, checkpointing, and recovery. The generated plan includes a physical operator DAG and a set of location constraints for each operator.

Runtime Context. The runtime context stores the cached `GS` tuple and maintains the Pregelix-specific implementations of the Hyracks extensible hooks to customize buffer, file, and index management.

3.5.8 Discussion

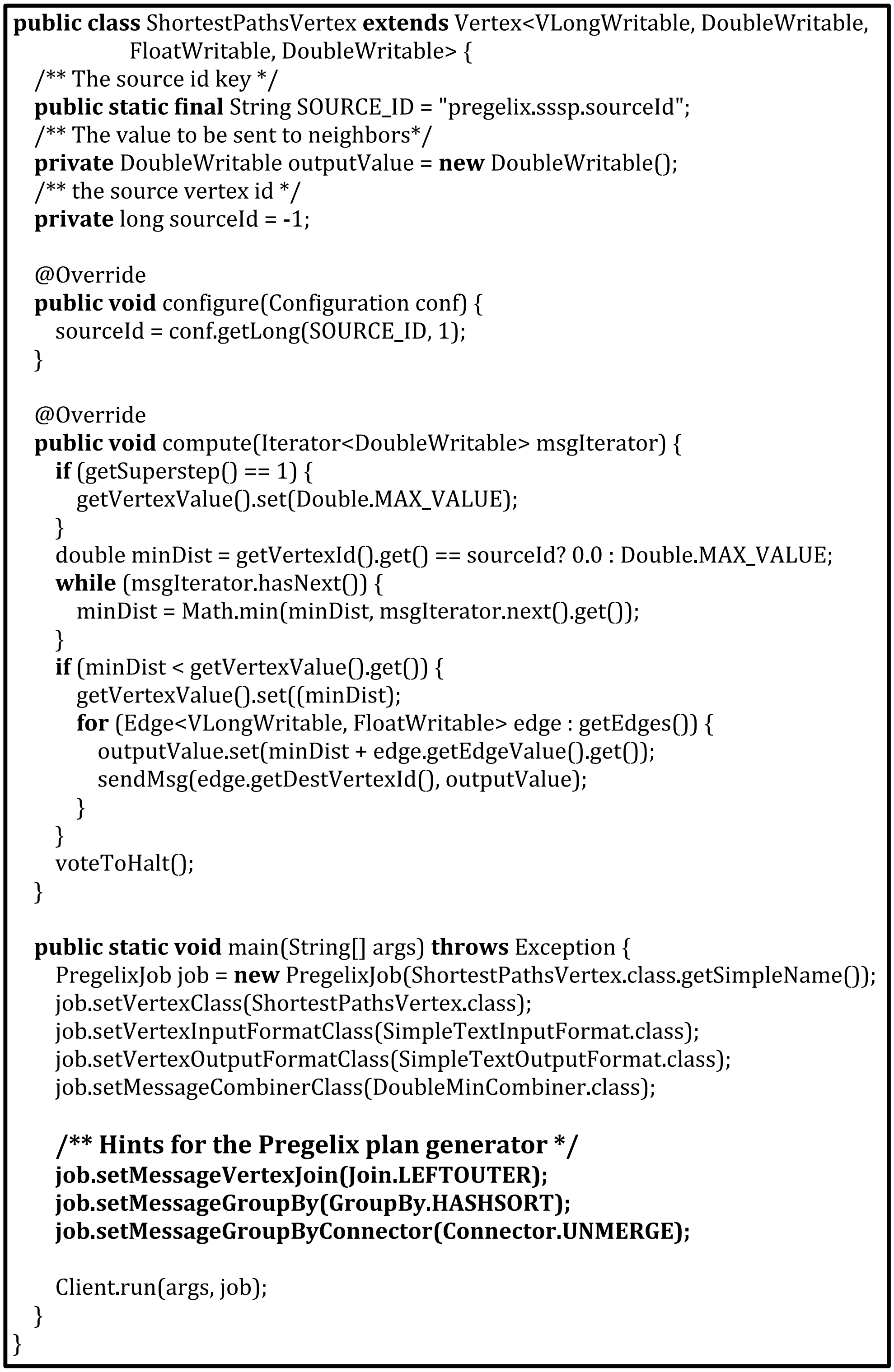
Let us close this section by revisiting the issues and opportunities presented in Section 3.2.3 and evaluating their implications in Pregelix:

- Out-of-core Support. All the data processing operators as well as access methods we use have out-of-core support, which allows the physical query plans on top to be able to run disk-based workloads as well as multi-user workloads while retaining good in-memory processing performance.
- Physical Flexibility. The current physical choices spanning vertex storage (two options), message delivery (two alternatives), and message combination (four strategies) allow Pregelix to have sixteen ($2 \times 2 \times 4$) tailored executions for different kinds of datasets, graph algorithms, and clusters.
- Software Simplicity. The implementations of all the described functionalities in this section leverage existing operator, connector, and access method libraries provided by Hyracks. Pregelix does not involve modifications to the Hyracks runtime.

3.6 Pregelix Case Studies

In this section, we briefly enumerate several Pregelix use cases, including a built-in graph algorithm library, a study of graph connectivity problems, and research on parallel genome assembly.

The Pregelix Built-in Library. The Pregelix software distribution comes with a library that includes several graph algorithms such as PageRank, single source shortest paths, connected components, reachability query, triangle counting, maximal cliques, and random-walk-based graph sampling. Figure 3.9 shows the single source shortest paths implementation



```
public class ShortestPathsVertex extends Vertex<VLongWritable, DoubleWritable,
            FloatWritable, DoubleWritable> {
  /** The source id key */
  public static final String SOURCE_ID = "pregelix.sssp.sourceId";
  /** The value to be sent to neighbors*/
  private DoubleWritable outputValue = new DoubleWritable();
  /** the source vertex id */
  private long sourceId = -1;

  @Override
  public void configure(Configuration conf) {
    sourceId = conf.getLong(SOURCE_ID, 1);
  }

  @Override
  public void compute(Iterator<DoubleWritable> msgIterator) {
    if (getSuperstep() == 1) {
      getVertexValue().set(Double.MAX_VALUE);
    }
    double minDist = getVertexId().get() == sourceId? 0.0 : Double.MAX_VALUE;
    while (msgIterator.hasNext()) {
      minDist = Math.min(minDist, msgIterator.next().get());
    }
    if (minDist < getVertexValue().get()) {
      getVertexValue().set((minDist);
      for (Edge<VLongWritable, FloatWritable> edge : getEdges()) {
        outputValue.set(minDist + edge.getEdgeValue().get());
        sendMsg(edge.getDestVertexId(), outputValue);
      }
    }
    voteToHalt();
  }

  public static void main(String[] args) throws Exception {
    PregelixJob job = new PregelixJob(ShortestPathsVertex.class.getSimpleName());
    job.setVertexClass(ShortestPathsVertex.class);
    job.setVertexInputFormatClass(SimpleTextInputFormat.class);
    job.setVertexOutputFormatClass(SimpleTextOutputFormat.class);
    job.setMessageCombinerClass(DoubleMinCombiner.class);

    /** Hints for the Pregelix plan generator */
    job.setMessageVertexJoin(Join.LEFTOUTER);
    job.setMessageGroupBy(GroupBy.HASHSORT);
    job.setMessageGroupByConnector(Connector.UNMERGE);

    Client.run(args, job);
  }
}
```

Figure 3.9: The implementation of the single source shortest paths algorithm on Pregelix.

on Pregelix, where hints for the join, group-by, and connector choices are set in the `main` function. Inside `compute`, the method calls to set a vertex value and to send a message internally generate output tuples for the corresponding dataflows.

Graph Connectivity Problems. Using Pregelix, a graph analytics research group in Hong Kong has implemented several graph algorithm building blocks such as BFS (breath first search) spanning tree, Euler tour, list ranking, and pre/post-ordering. These building blocks have been used to develop advanced graph algorithms such as bi-connected components for undirected graphs (e.g., road networks) and strongly connected components for directed graphs (e.g., the Twitter follower network) [106] . The group also scale-tested all of their algorithms on a 60 machine cluster with 480 cores and 240 disks, using Pregelix as the infrastructure.

Genome Assembly. Genomix [8] is a data-parallel genome assembler built on top of Pregelix. It first constructs a (very large) De Bruijn graph [109] from billions of genome reads, and then (a) cleans the graph with a set of pre-defined subgraph patterns (described in [109]) and (b) merges available single paths into vertices iteratively until all vertices can be merged to a single (gigantic) genome sequence. Pregelix's support for the addition and removal of vertices is heavily used in this use case.

3.7 Experiments

This section compares Pregelix with several other popular parallel graph processing systems, including Giraph [9], Hama [11], GraphLab [73], and GraphX [97]. Our comparisons cover execution time (Section 3.7.2), scalability (Section 3.7.3), throughput (Section 3.7.4), plan flexibility (Section 3.7.5), and software simplicity (Section 3.7.6). We conclude this section by summerizing our experimental results (Section 3.7.7).

3.7.1 Experimental Setup

We ran the experiments detailed here on a 32-node Linux IBM x3650 cluster with one additional master machine of the same configuration. Nodes are connected with a Gigabit Ethernet switch. Each node has one Intel Xeon processor E5520 2.26GHz with four cores, 8GB of RAM, and two 1TB, 7.2K RPM hard disks.

In our experiments, we leverage two real-world graph-based datasets. The first is the Webmap dataset [105] taken from a crawl of the web in the year 2002. The second is the BTC dataset [49], which is a undirected semantic graph converted from the original Billion Triple Challenge 2009 RDF dataset [1]. Table 3.3 (Webmap) and Table 3.4 (BTC) show statistics for these graph datasets, including the full datasets as well as several down-samples and scale-ups[7] that we use in our experiments.

Our evaluation examines the platforms' performance characteristics of three algorithms: PageRank [83], SSSP (single source shortest paths) [56], and CC (connected components) [56]. On Pregelix and Giraph, the graph algorithms were coded in Java and all their source code can be found in the Pregelix codebase[8]. The implementations of the three algorithms on Hama, GraphLab, and GraphX are directly borrowed from their builtin examples. The Pregelix default plan, which uses index full outer join, sort-based group-by, an m-to-n hash partitioning connector, and B-tree vertex storage, is used in Sections 3.7.2, 3.7.3, and 3.7.4. Pregelix's default maximum buffer cache size for access methods is set to $\frac{1}{4}$ the physical RAM size, and its default maximum allowed buffer for each group-by operator instance is set to 64MB. These two default Pregelix memory settings are used in all the experiments. For the local file system for Pregelix, we use the ext3 file system; for the distributed file system, we use HDFS version 1.0.4. In all experiments, we use Giraph trunk version (git revision 770

[7] We used a random walk graph sampler built on top of Pregelix to create scaled-down Webmap sample graphs of different sizes. To scale up the BTC data size, we deeply copied the original graph data and renumbered the duplicate vertices with a new set of identifiers.

[8] https://code.google.com/p/hyracks/source/browse/pregelix

Name	**Size**	**#Vertices**	**#Edges**	**Avg. Degree**
Large	71.82GB	1,413,511,390	8,050,112,169	5.69
Medium	31.78GB	709,673,622	2,947,603,924	4.15
Small	14.05GB	143,060,913	1,470,129,872	10.27
X-Small	9.99GB	75,605,388	1,082,093,483	14.31
Tiny	2.93GB	25,370,077	318,823,779	12.02

Table 3.3: The Webmap dataset (Large) and its samples.

Name	**Size**	**#Vertices**	**#Edges**	**Avg. Degree**
Large	66.48GB	690,621,916	6,177,086,016	8.94
Medium	49.86GB	517,966,437	4,632,814,512	8.94
Small	33.24GB	345,310,958	3,088,543,008	8.94
X-Small	16.62GB	172,655,479	1,544,271,504	8.94
Tiny	7.04GB	107,706,280	607,509,766	5.64

Table 3.4: The BTC dataset (X-Small) and its samples/scale-ups.

in May 2014), Hama version 0.6.4, GraphLab version 2.2 (PowerGraph), and Spark [108] version 0.9.1 for GraphX. Our GraphLab setting has been confirmed by its primary author. We tried our best to let each system use all the CPU cores and all available RAM on each worker machine.

3.7.2 Execution Time

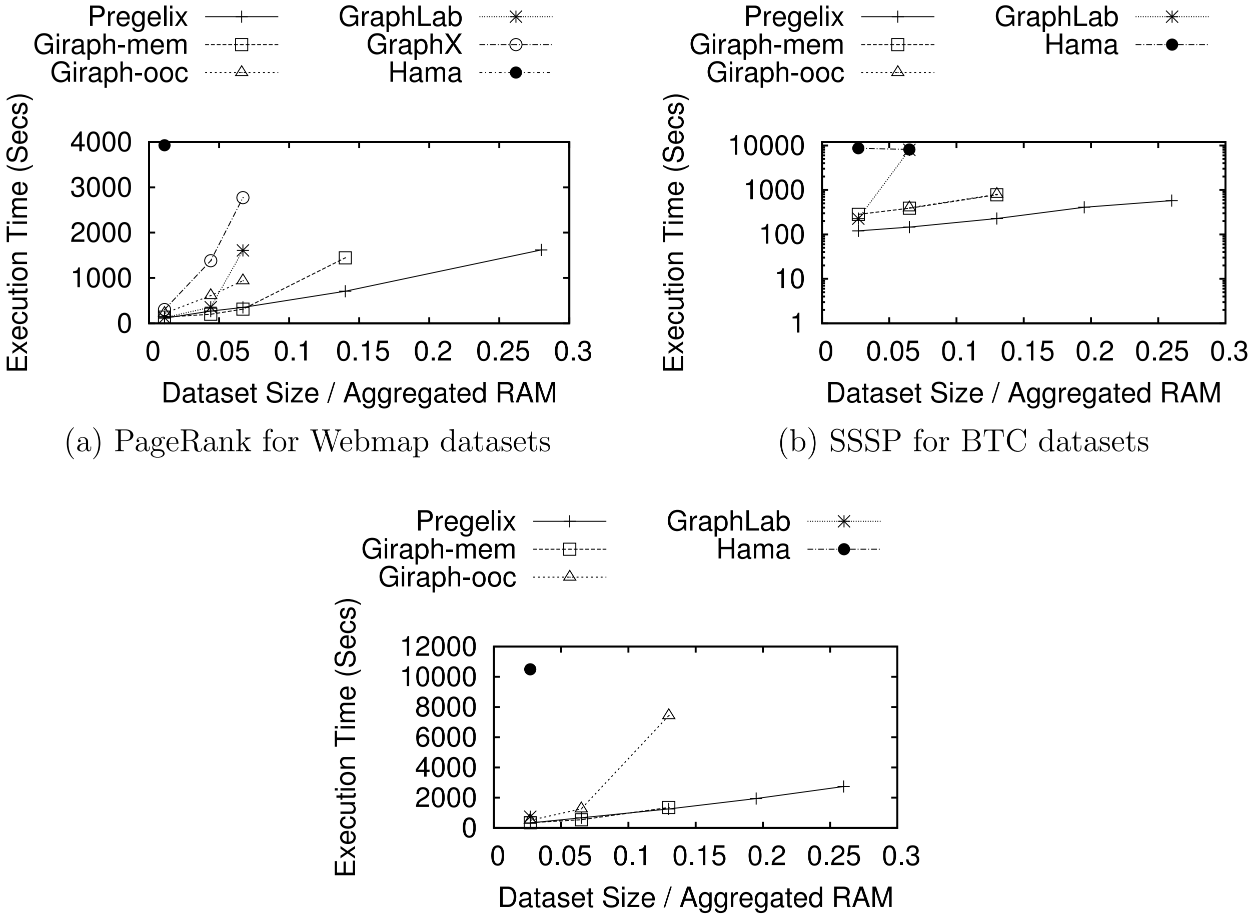
In this experiment, we evaluate the execution times of all systems by running each of the three algorithms on the 32-machine cluster. As the input data for PageRank we use the Webmap dataset because PageRank is designed for ranking web pages, and for the SSSP and CC algorithms we use the BTC dataset. Since a Giraph user needs to explicitly specify apriori whether a job is in-memory or out-of-core, we measure both of these settings (labeled Giraph-mem and Giraph-ooc, respectively) for Giraph jobs regardless of the RAM size.

Figure 3.10 plots the resulting *overall* Pregel job execution times for the different sized datasets, and Figure 3.11 shows the average *per-iteration* execution time for all iterations.

In both figures, the x-axis is the input dataset size relative to the cluster's aggregated RAM size, and the y-axis is the execution time. (Note that the volume of exchanged messages can exhaust memory even if the initial input graph dataset can fit into memory.)

Figure 3.10 and Figure 3.11 show that while Pregelix scales to out-of-core workloads, Giraph fails to run the three algorithms once the relative dataset size exceeds 0.15, even with its out-of-core setting enabled. Figures 3.10(a)(c) and 3.11(a)(c) show that when the computation has sufficient memory, Pregelix offers comparable execution time to Giraph for message-intensive workloads such as PageRank and CC. For PageRank, Pregelix runs up to 2× slower than Giraph on relatively small datasets, e.g., Webmap-X-Small. For CC, Pregelix runs upto 1.7× faster than Giraph for relatively small datasets like BTC-Small. Figure 3.10(b) and Figure 3.11(b) further demonstrate that the Pregelix default plan offers 3.5× overall speedup and 7× per-iteration speedup over Giraph for message-sparse workloads like SSSP even for relatively small datasets. All sub-figures in Figure 3.10 and Figure 3.11 show that for in-memory workloads (when the relative dataset size is less than 0.15), Giraph has steeper (worse) size-scaling curves than Pregelix.

Compared to Giraph, GraphLab, GraphX, and Hama start failing on even smaller datasets, with even steeper size-scaling curves. GraphLab has the best average per-iteration execution time on small datasets (e.g., up to 5× faster than Pregelix and up to 12× faster than Giraph, on BTC-Tiny), but performs worse than Giraph and Pregelix on larger datasets (e.g., up to 24× slower than Pregelix and up to 6× slower than Giraph, on BTC-X-Small). GraphX cannot successfully load the BTC-Tiny dataset on the 32-machine cluster, therefore its results for SSSP and CC are missing.



(a) PageRank for Webmap datasets

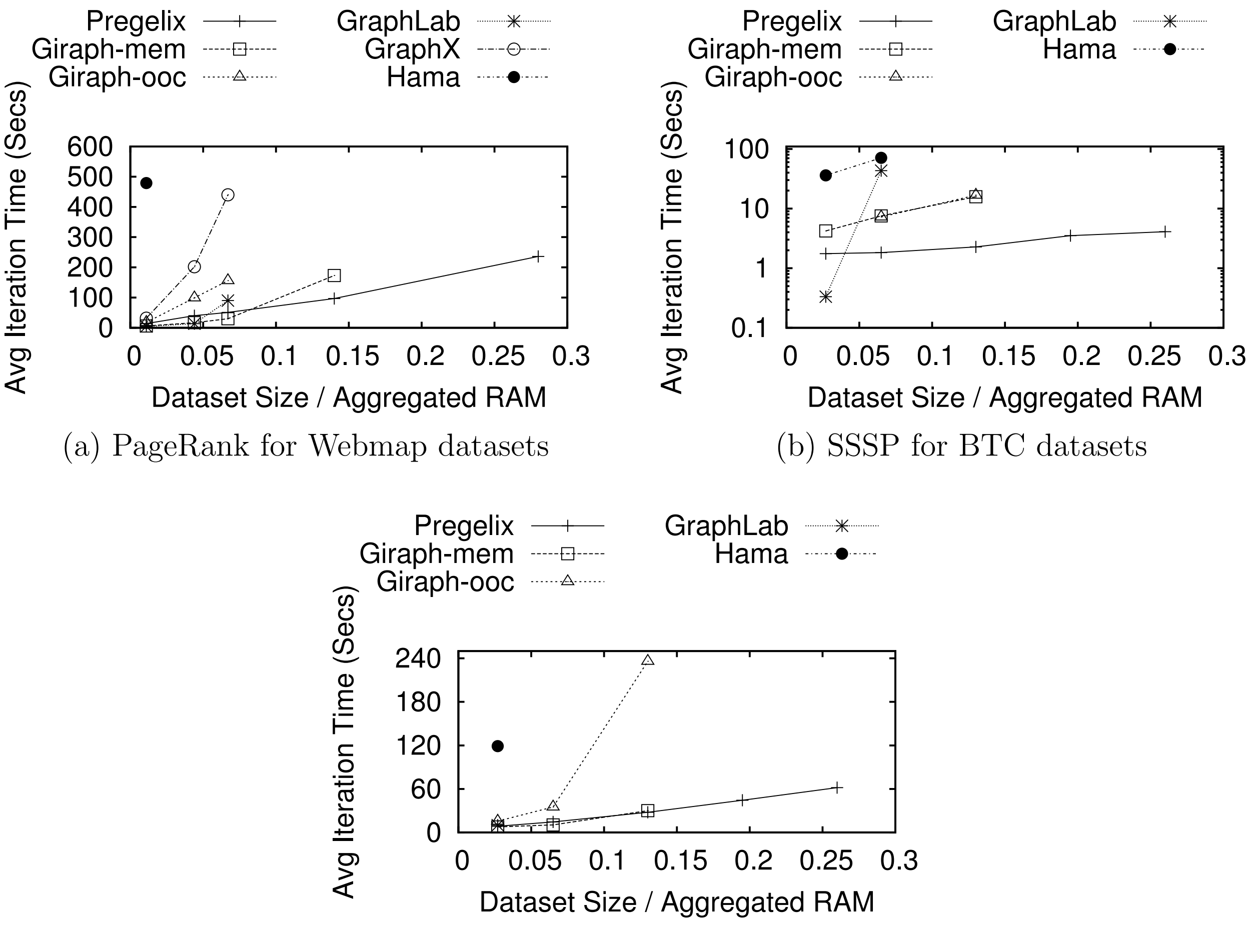
(b) SSSP for BTC datasets

(c) CC for BTC datasets

Figure 3.10: Overall execution time (32-machine cluster). Neither Giraph-mem nor Giraph-ooc can work properly when the ratio of dataset size to the aggregated RAM size exceeds 0.15; GraphLab starts failing when the ratio of dataset size to the aggregated RAM size exceeds 0.07; Hama fails on even smaller datasets; GraphX fails to load the smallest BTC dataset sample BTC-Tiny.

3.7.3 System Scalability

Our system scalability experiments run the different systems on varying sized clusters for each of the different dataset sizes. Figure 3.12(a) plots the parallel speedup for PageRank on Pregelix going from 8 machines to 32 machines. The x-axis is the number of machines, and the y-axis is the average per-iteration execution time relative to the time on 8 machines. As the number of machines increases, the message combiner for PageRank becomes less



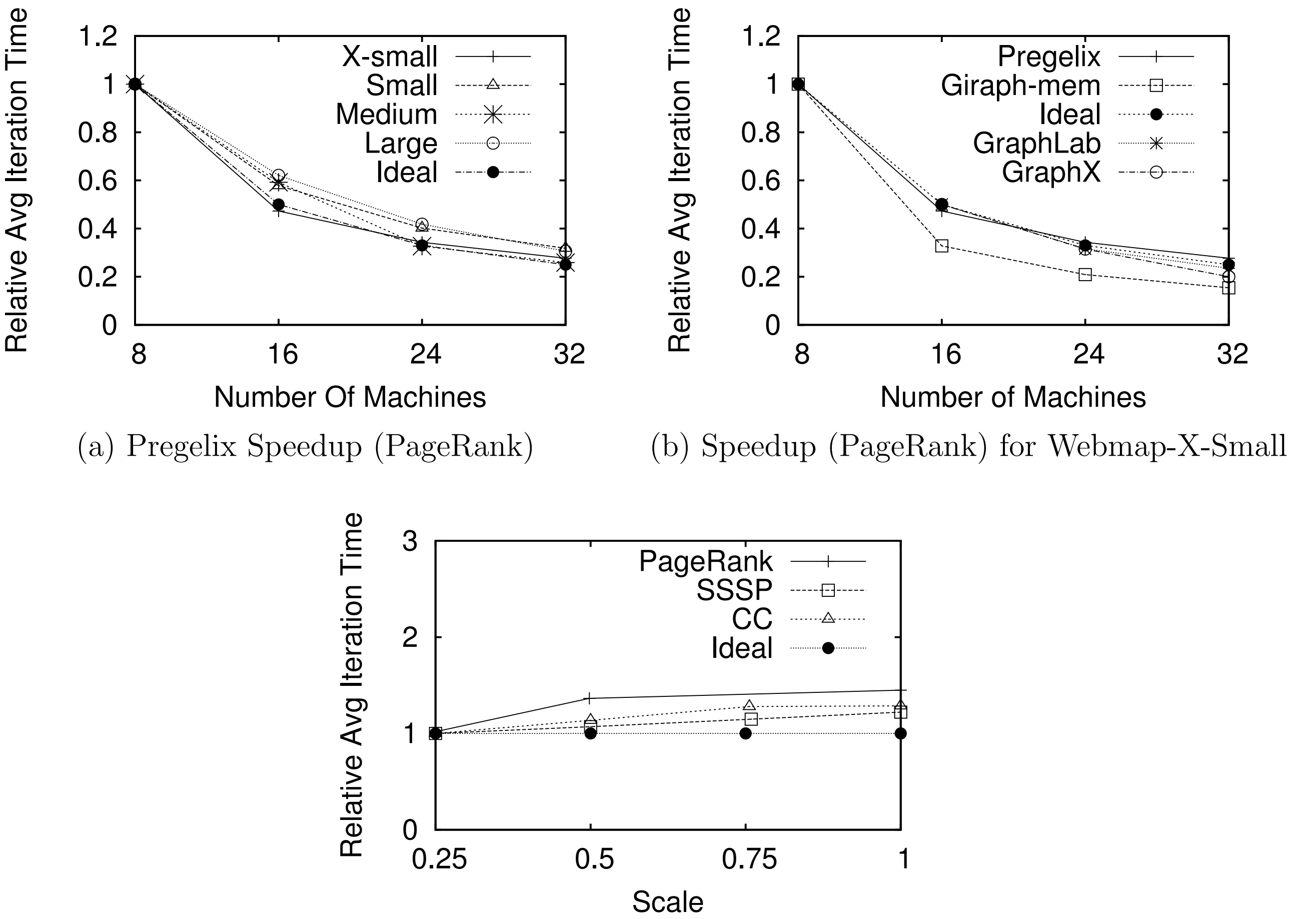
(a) PageRank for Webmap datasets

(b) SSSP for BTC datasets

(c) CC for BTC datasets

Figure 3.11: Average iteration execution time (32-machine cluster).

effective and hence the total volume of data transferred through the network gets larger, though the CPU load of each individual machine drops. Therefore, in Figure 3.12(a), the parallel speedups are close to but slightly worse than the "ideal" case in which there are no message overheads among machines. For the other systems, the PageRank implementations for GraphLab and GraphX fail to run Webmap samples beyond the 9.99GB case when the number of machines is 16; Giraph has the same issue when the number of machines is 8. Thus, we were only able to compare the parallel PageRank speedups of Giraph, GraphLab, GraphX, and Pregelix with the Webmap-X-Small dataset. The results of this small data case are in Figure 3.12(b); Hama is not included because it cannot run even the Webmap-



(a) Pregelix Speedup (PageRank)

(b) Speedup (PageRank) for Webmap-X-Small

(c) Pregelix Scaleup (PageRank, SSSP, CC)

Figure 3.12: Scalability (run on 8-machine, 16-machine, 24-machine, 32-machine clusters).

X-Small dataset on our cluster configurations. The parallel speedup of Pregelix is very close to the ideal line, while Giraph, GraphLab, and GraphX exhibit even better speedups than the ideal. The apparent super-linear parallel speedups of Giraph, GraphLab, and GraphX are consistent with the fact that they all perform super-linearly worse when the volume of data assigned to a slave machine increases, as can be seen in Figures 3.10 and 3.11.

Figure 3.12(c) shows the parallel scale up of the three algorithms for Pregelix. Giraph, GraphLab, GraphX, and Hama results are not shown because they could not run all these cases. In this figure, the x-axis is the ratio of the sampled (or scaled) dataset size over the largest (Webmap-Large or BTC-Large) dataset size. The number of machines is proportional to this ratio and 32 machines are used for scale 1.0. The y-axis is the average per-iteration execution time relative to the time at the smallest scale. In the ideal case, the y-axis value would stay at 1.0 for all the scales, while in reality, the three Pregel graph algorithms all have network communication and thus cannot achieve the ideal. The SSSP algorithm sends fewer messages than the other two algorithms, so it is the closest to the ideal case.

3.7.4 Throughput

In the current version of GraphLab, Hama, and Pregelix, each submitted job is executed immediately, regardless of the current activity level on the cluster. Giraph leverages Hadoop's admission control; for the purpose of testing concurrent workloads, we let the number of Hadoop map task slots in each task tracker be 3. GraphX leverages the admission control of Spark, such that jobs will be executed sequentially if available resources cannot meet the overall requirements of concurrent jobs. In this experiment, we compare the job throughput of all the systems by submitting jobs concurrently. We ran PageRank jobs on the 32-machine cluster using four different samples of the Webmap dataset (X-Small, Small, Medium, and Large) with various levels of job concurrency. Figure 3.13 reports how the number of completed jobs per hour (**jph**) changes with the number of concurrent jobs. The results for the four Webmap samples represent four different cases respectively:

- Figure 3.13(a) uses Webmap-X-Small. Moving from serial job execution to concurrent job execution, data processing remains in-memory but the CPU resources go from dedicated to shared. In this case, Pregelix achieves a higher jph when there are two or three concurrent jobs than when job execution is serial.
- Figure 3.13(b) uses Webmap-Small. In this case, serial job execution does in-memory processing, but concurrent job execution introduces a small amount of disk I/O due to spilling. When two jobs run concurrently, each job incurs about 1GB of I/O; when three

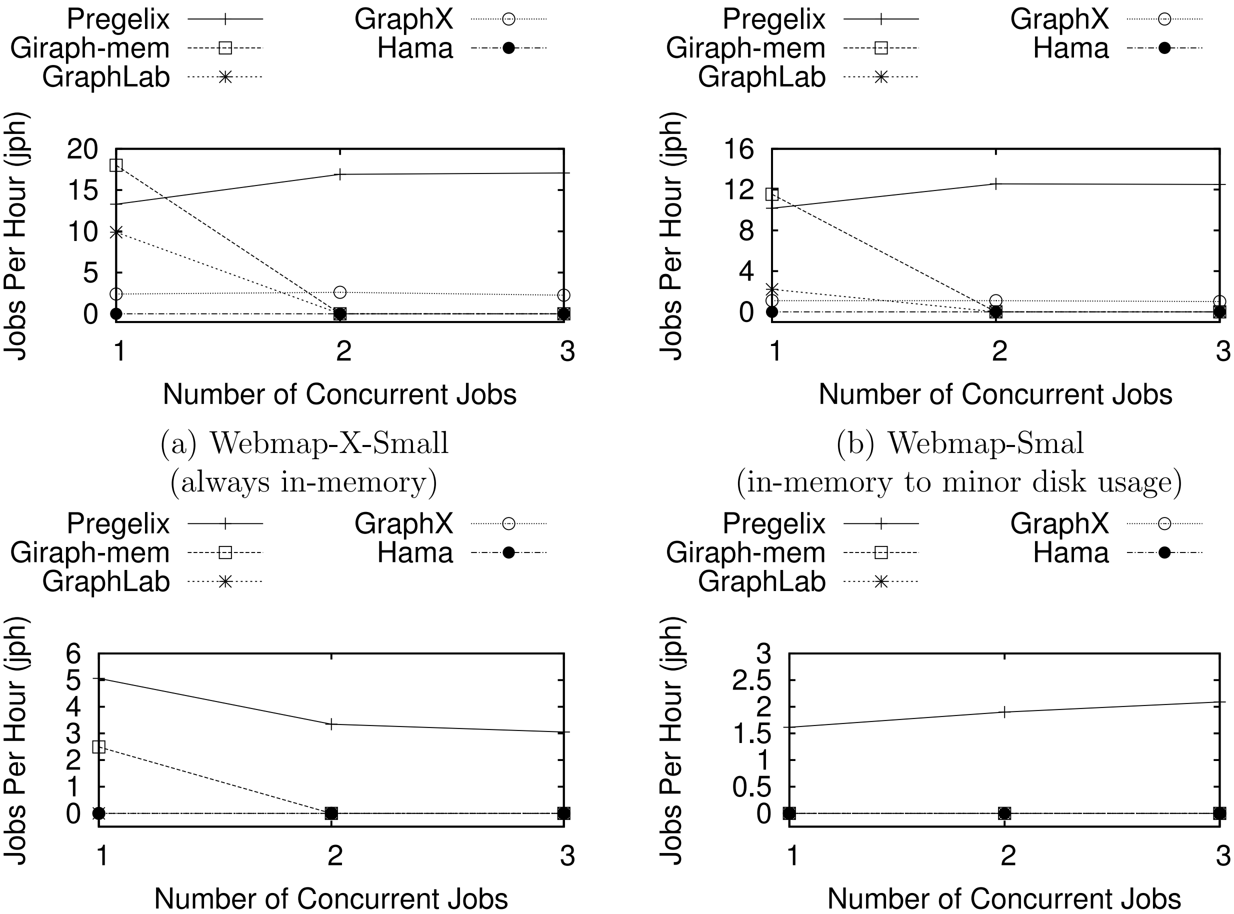
jobs run concurrently, each does about 2.7GB of I/O. In this situation, still, the Pregelix jph is higher in concurrent execution than in serial execution.

- Figure 3.13(c) uses Webmap-Medium. In this case, serial job execution allows for in-memory processing, but allowing concurrent execution exhausts memory and causes a significant amount of I/O for each individual job. For example, when two jobs run concurrently, each job incurs about 10GB of I/O; when three jobs run concurrently, each job does about 27GB of I/O. In this case, jph drops significantly at the boundary where I/O significantly comes into the picture. The point with two concurrent jobs is such a boundary for Pregelix in Figure 3.13(c).
- Figure 3.13(d) uses the full Webmap (Webmap-Large). In this case, processing is always disk-based regardless of the concurrency level. For this case, Pregelix jph once again increases with the increased level of concurrency; this is because the CPU utilization is increased (by about 20% to 30%) with added concurrency.

These results suggest that it would be worthwhile to develop intelligent job admission control policies to make sure that Pregelix runs with the highest possible throughput all the time in our future work. In our experiments, the Spark scheduler for GraphX always runs concurrent jobs sequentially due to the lack of memory and CPU resources. Giraph, GraphLab, and Hama all failed to support concurrent jobs in our experiments for all four cases due to limitations regarding memory management and out-of-core support; they each need additional work to operate in this region.

3.7.5 Plan Flexibility

In our final experiment, we compare several different physical plan choices in Pregelix to demonstrate their usefulness. We ran the two join plans (described in Section 3.5.3.2) for the three Pregel graph algorithms. Figure 3.14 shows the results. For message-sparse



(a) Webmap-X-Small
(always in-memory)

(b) Webmap-Smal
(in-memory to minor disk usage)

(c) Webmap-Medium
(in-memory to disk-based)

(d) Webmap-Large
(always disk-based)

Figure 3.13: Throughput (multiple PageRank jobs are executed in the 32-machine cluster with different sized datasets).

algorithms like SSSP (Figure 3.14(a)), the left outer join Pregelix plan is much faster than the (default) full outer join plan. However, for message-intensive algorithms like PageRank (Figure 3.14(b)), the full outer join plan is the winner. This is because although the probe-based left outer join can avoid a sequential index scan, it needs to search the index from the root node every time; this is not worthwhile if most data in the leaf nodes will be qualified as join results. The CC algorithm's execution starts with many messages, but the message volume decreases significantly in its last few supersteps, and hence the two join plans result in similar performance (Figure 3.14(c)). Figure 3.15 revisits the relative performance of the systems by comparing Pregelix's left outer join plan performance against the other systems. As shown in the figure, SSSP on Pregelix can be up to 15× faster than on Giraph and up to 35× faster than on GraphLab for the average per-iteration execution time when Pregelix is run with its left outer join plan.

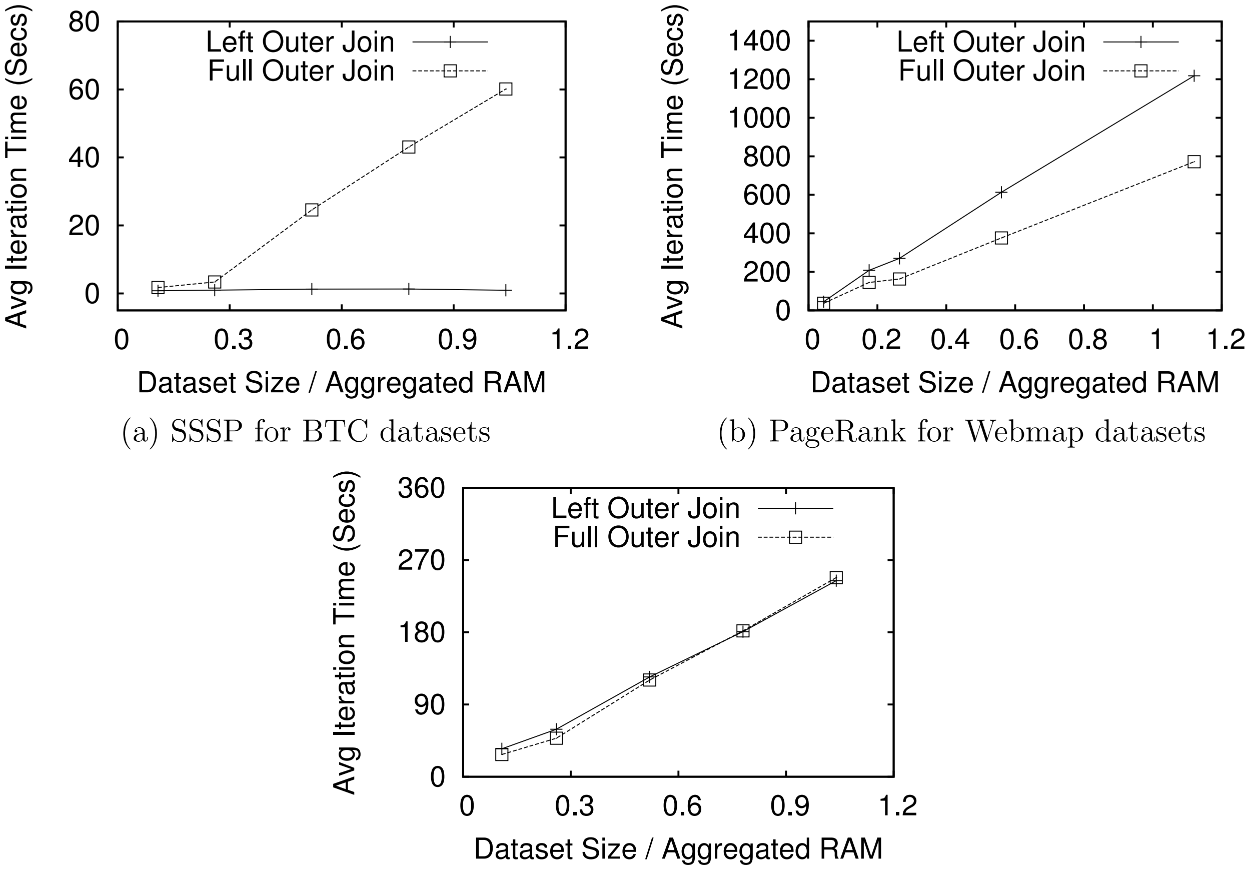
In addition to the experiments presented here, an earlier technical report [44] measured the performance difference introduced by the two different Pregelix data redistribution policies (as described in Section 3.5.3.1) for combining messages on a 146-machine cluster in Yahoo! Research. Figure 9 in the report [44] shows that the m-to-n hash partitioning merging connector can lead to slightly faster executions on small clusters, but merging input streams at the receiver side needs to selectively wait for data to arrive from specific senders as dictated by the priority queue, and hence it becomes slower on larger clusters. The tradeoffs seen here and in [44] for different physical choices are evidence that an optimizer is ultimately essential to identify the best physical plan to use in order to efficiently execute Pregel programs.

3.7.6 Software Simplicity

To highlight the benefit of building a Pregel implementation on top of an existing dataflow runtime, as opposed to writing one from scratch, we can count lines of code for both the Pregelix and Giraph systems' core modules (excluding their test code and comments). The Giraph-core module, which implements the Giraph infrastructure, contains 32,197 lines of code. Its counterpart in Pregelix contains just 8,514 lines of code.

3.7.7 Summary

Our experimental results show that Pregelix can perform comparably to Giraph for memory-resident message-intensive workloads (like PageRank), can outperform Giraph by over an or-



(a) SSSP for BTC datasets

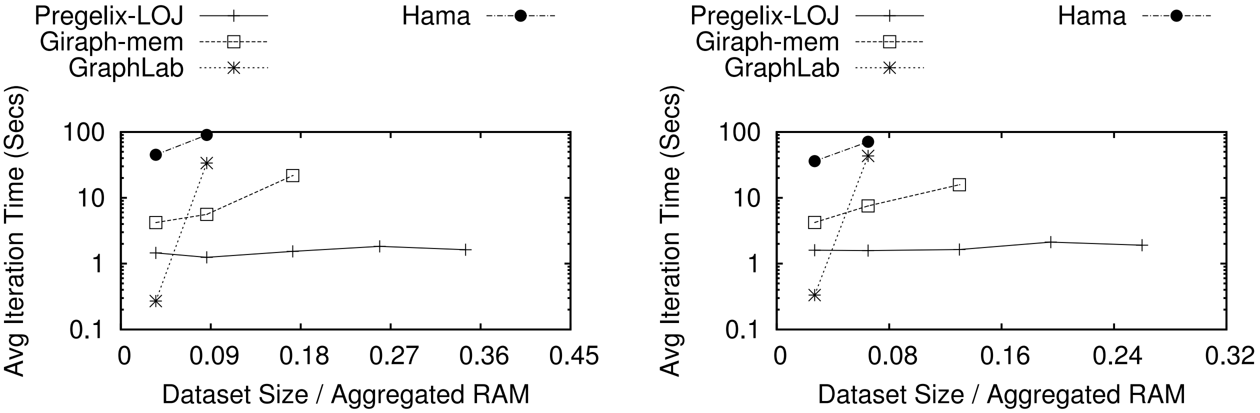
(b) PageRank for Webmap datasets

(c) CC for BTC datasets

Figure 3.14: Index full outer join vs. index left outer join (run on an 8 machine cluster) for Pregelix.

der of magnitude for memory-resident message-sparse workloads (like single source shortest paths), can scale to larger datasets, can sustain multi-user workloads, and also can outperform GraphLab, GraphX, and Hama by over an order of magnitude on large datasets for various workloads. In addition to the numbers reported in this chapter, an early technical report [44] gave speedup and scale-up results for the first alpha release of Pregelix on a 146-machine Yahoo! Research cluster in March 2012; that study first showed us how well the Pregelix architectural approach can scale[9].

[9] Unfortunately, our Yahoo! collaborators have since left the company, so we no longer have access to a cluster of that scale.



(a) 24-Machine Cluster

(b) 32-Machine Cluster

Figure 3.15: Pregelix left outer join plan vs. other systems (SSSP on BTC datasets). GraphX fails to load the smallest dataset.

3.8 Related Work

The Pregelix system is related to or built upon previous works from three areas.

Parallel data management systems such as Gamma [53], Teradata [21], and GRACE [59] applied partitioned-parallel processing to SQL query processing over two decades ago. The introduction of Google's MapReduce system [52], based on similar principles, led to the recent flurry of work in MapReduce-based data-intensive computing. Systems like Dryad [70], Hyracks [42], and Nephele [38] have successfully made the case for supporting a richer set of data operators beyond map and reduce as well as a richer set of data communication patterns. REX [76] integrated user-defined delta functions into SQL to support arbitrary recursions and built stratified parallel evaluations for recursions. The Stratosphere project also proposed an incremental iteration abstraction [57] using working set management and integrated it with parallel dataflows and job placement. The lessons and experiences from all of these systems provided a solid foundation for the Pregelix system.

Big Graph processing platforms such as Pregel [75], Giraph [9], and Hama [11] have been built to provide vertex-oriented message-passing-based programming abstractions for distributed graph algorithms to run on shared-nothing clusters. Sedge [107] proposed an efficient advanced partitioning scheme to minimize inter-machine communications for Pregel computations. Surfer [48] is a Pregel-like prototype using advanced bandwidth-aware graph partitioning to minimize the network traffic in processing graphs. In seeming contradiction to these favorable results on the efficacy of smart partitioning, literature [69] found that basic hash partitioning works better because of the resulting balanced load and the low partitioning overhead. We seem to be seeing the same with respect to GraphLab, i.e., the pain is not being repaid in performance gain. GPS [86] optimizes Pregel computations by dynamically repartitioning vertices based on message patterns and by splitting high-degree vertices across all worker machines. Giraph++ [93] enhanced Giraph with a richer set of APIs for user-defined partitioning functions so that communication within a single partition can be bypassed. GraphX [97] provides a programming abstraction called Resident Distributed Graphs (RDGs) to simplify graph loading, construction, transformation, and computations, on top of which Pregel can be easily implemented. Different from Pregel, GraphLab [73] provides a vertex-update-based programming abstraction and supports an asynchronous model to increase the level of pipelined parallelism. Trinity [90] is a distributed graph processing engine built on top of a distributed in-memory key-value store to support both online and offline graph processing; it optimizes message-passing in vertex-centric computations for the case where a vertex sends messages to a fixed set of vertices. Our work on Pregelix is largely orthogonal to these systems and their contributions because it looks at Pregel at a lower architectural level, aiming at better out-of-core support, plan flexibility, and software simplicity.

Iterative extensions to MapReduce like HaLoop [45] and PrIter [110] were the first to extend MapReduce with looping constructs. HaLoop hardcodes a sticky scheduling policy (similar to the one adopted here in Pregelix and to the one in Stratosphere [57]) into the Hadoop task scheduler so as to introduce a caching ability for iterative analytics. PrIter uses a key-value storage layer to manage its intermediate MapReduce state, and it also exposes user-defined policies that can prioritize certain data to promote fast algorithmic convergence. However, those extensions still constrain computations to the MapReduce model, while Pregelix explores more flexible scheduling mechanisms, storage options, operators, and several forms of data redistribution (allowed by Hyracks) to optimize a given Pregel algorithm's computation time.

3.9 Summary

This chapter has presented the design, implementation, early use cases, and evaluation of Pregelix, a new dataflow-based Pregel-like system built on top of the Hyracks parallel dataflow engine. Pregelix combines the Pregel API from the systems world with data-parallel query evaluation techniques from the database world in support of Big Graph Analytics. This combination leads to effective and transparent out-of-core support, scalability, and throughput, as well as increased software simplicity and physical flexibility. To the best of our knowledge, Pregelix is the only open source Pregel-like system that scales to out-of-core workloads efficiently, can sustain multi-user workloads, and allows runtime flexibility. This sort of architecture and methodology could be adopted by parallel data warehouse vendors (such as Teradata [21], Pivotal [18], or Vertica [26]) to build Big Graph processing infrastructures on top of their existing query execution engines. Last but not least, we have made several stable releases of the Pregelix system (http://pregelix.ics.uci.edu) in open source form since the year 2012 for use by the Big Data research community, and we invite others to download and try the system.

This chapter has primarily focused on processing large amounts of explicit graph data, but in real-world, graph topologies are often hidden in large volumes of tabular data which seem irrelevant to graphs. In the next chapter, the Pregelix system we have described in this chapter will be used as a build block for adding more complex graph analytical pipeline support in AsterixDB.

Chapter 4

Rich(er) Graph Analytics in AsterixDB

4.1 Overview

Chapter 3 has proposed and evaluated a dataflow approach for building a scalable Big Graph analytics platform. However, Big Graph analytics are not only about graphs. Many real-world applications do not have a pre-loaded graph dataset for analysis, and instead, graphs are often constructed by querying the data that are dynamically ingested into an enterprise data lake [4], i.e., a storage repository (e.g., HDFS [10]) that holds a vast amount of raw data. Figure 4.1 shows a dataflow for rich(er) forms of graph analytics, i.e., end-to-end analytical pipelines spanning from the raw data to the final mined insights. As the figure indicates, various data from web, mobile, and IoT (internet of things) applications are continuously ingested into a data lake. In the meantime, data scientists run SQL [20] queries using typical SQL-on-Hadoop systems (e.g., Hive [92], Impala [71], or Tez [36]) to extract graphs (e.g., in the adjacency list representation) of interests from the raw data, put the intermediate graph

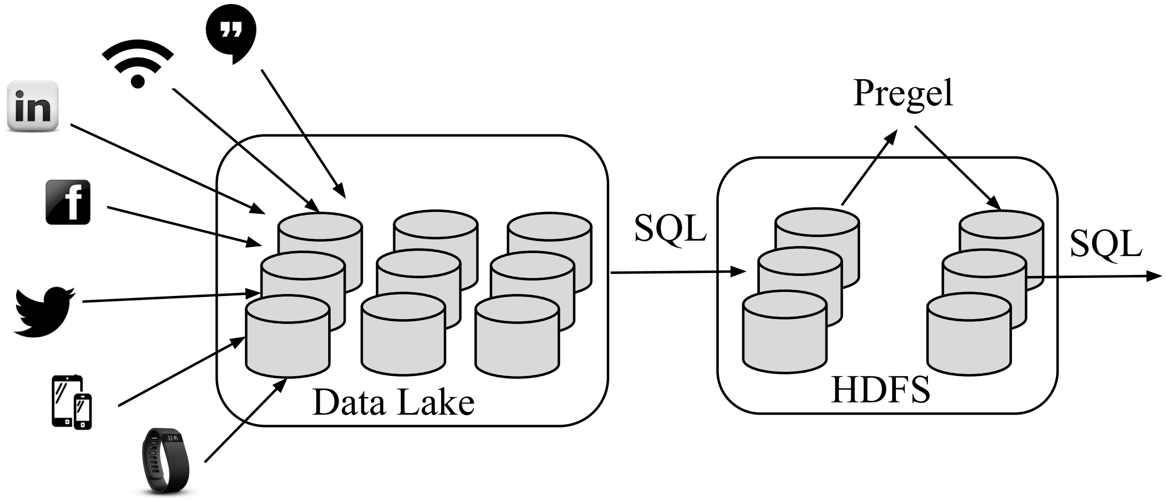


Figure 4.1: The current Big Graph analytics flow.

data onto HDFS, run distributed graph algorithms using typical graph analytical systems (e.g., Pregel [75], Giraph [9], GraphLab [73], or Pregelix [19]), and finally run another set of SQL queries over the graph computation results to generate user-readable reports [13]. In this process, a data scientist needs to figure out the physical locations of the intermediate results, manage their life-cycles, determine their formats, and write customized client scripts to glue SQL-on-Hadoop systems and graph analytical systems together through HDFS. All these tedious ETL (extract, transform, load) tasks draw the data scientist's energy from thinking of the analytical task at a logical level.

Let us use a motivating example to illustrate the problem. The example analytical task we use is to find the top 50 most influential people on Twitter, using their tweets, with Hive [92] and Giraph [9]. We first create an external table in Hive for the tweet dataset as follows:

```
CREATE EXTERNAL TABLE TwitterMessage
(
   tweetid: int64,
   user: string,
   sender_location: point,
   send_time: datetime,
   reply_from: int64,
   retweet_from: int64,
   referred_topics: array<string>,
   message_text: string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘,’ COLLECTION ITEMS TERMINATED BY ‘%’
LINES TERMINATED BY ’\n’  STORED AS TEXTFILE LOCATION ‘/tw_message’;
```

In this analysis, we treat each reply or retweet as a graph vertex in which either the `reply_from` field or the `retweet_from` field is extracted as an outgoing edge. If a given tweet message gets replied to more or retweeted more, we will say that the message has a larger impact. Thus, we create an intermediate table for such a graph and run a SQL query to populate it, as follows.

```
CREATE EXTERNAL TABLE MsgGraph
(vertexid int64, double rank, dests array<int64>)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘,’
COLLECTION ITEMS TERMINATED BY ‘%’  LINES TERMINATED BY ’\n’
STORED AS TEXTFILE  LOCATION ‘/msggraph’;

INSERT OVERWRITE TABLE MsgGraph
SELECT
        T.tweetid,
        1.0/10000000000.0,  /* Assign a very small initial rank to every tweet. */
        CASE
        /* Each selected tweet can only be one of the three cases */
        WHEN T.reply_from>=0 RETURN array(T.reply_from)
        ELSE RETURN array(T.forward_from)
        END CASE
FROM TwitterMessage AS T
WHERE T.reply_from>=0 OR T.forward_from>=0  /*Only replies and retweets are retained.*/
```

Note that, the `insert` statement only adds replies and retweets into table `MsgGraph` without their original tweets — this is because a Pregel runtime (described in Section 3.3) will add those original tweets (vertexes) if there are messages sent to them in a Pregel computation superstep. Before running the above query, we should make sure that the HDFS directory `/msggraph` is not used for other purposes. After loading the graph, we need to implement the InputFormat/OutputFormat interfaces to let Giraph be able to read the CSV (comma separated values) files in directory `/msggraph` and write result files in the form of CSV text as well. Given the InputFormat/OutputFormat classes as well as the input path (i.e., `/msggraph`) and output path (let us assume that the output path is `/results`) as the arguments, we next run a Giraph job to do PageRank [1]. From there, we run yet-anther SQL query to get the top influencers by considering the aggregated message rank values of each user:

```
CREATE EXTERNAL TABLE Result
(vertexid int64, rank double)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘,’ COLLECTION ITEMS TERMINATED BY ‘%’
LINES TERMINATED BY ’\n’  STORED AS TEXTFILE   LOCATION ‘/results ;

SELECT T.user, SUM(R.rank) as influence
FROM Result AS  R  JOIN TwitterMessage AS  T
ON R.vertexid = T.tweetid
GROUP BY T.user
ORDER BY influence DESC
LIMIT 50
```

Finally, we should decide to either erase the intermediate temporary directories such as `/msggraph` and `/results` or keep them for a while for other uses.

As we can see from this example, in order to do such a SQL-Pregel combined analysis, a data scientist has to take ownership of the following physical details that are logically irrelevant to the analytical task:

[1] In all the queries in this chapter, we set the “damping factor” of the PageRank algorithm [83] to be 1.0 to obtain the relative rank value of each vertex.

- the safety and lifecycle of intermediate files;
- the DDLs and parsers that understand the physical formats of the intermediate data;
- the script that glues the clients of different systems together.

To address these issues, in this chapter, we add Big Graph analytics support into AsterixDB, a Big Data management system for ingesting and querying semi-structured data [29]. We improve the user experience in the example by integrating a Pregelix job entrance into a declarative query language, i.e., AQL (AsterixDB Query Language), and by exposing an AsterixDB input/output converter API in Pregelix that separates out the physical details for the intermediate data. The resulting combination not only allows a data scientist to stay at the logical level for solving analytical problems, but it also enables interactive graph analytics by leveraging the secondary indexing capability of AsterixDB [30]. The contributions of this chapter include:

- We describe and add the support for temporary datasets, which are logging- and locking-free, into AsterixDB (Section 4.2).
- We design and build an external connector framework for Pregelix and implement an AsterixDB connector on top of the framework (Section 4.3).
- We add a Pregel job entrance capability in AQL (Section 4.4).
- We experimentally evaluate the efficiency of temporary datasets as well as the scaling properties of rich(er) graph analytical queries in AsterixDB (Section 4.5).

It is worth nothing that the resulting framework, while applied here to graph analytics, is more general. Other analytics platforms can (and will) also be lossely coupled with AsterixDB in this manner.

4.2 Temporary Datasets In AsterixDB

This section describes the motivation, design, and implementation of temporary datasets in AsterixDB. We first revisit storage management in AsterixDB (Section 4.2.1), then we discuss the needs of temporary datasets, and finally we dive into the details of temporary datasets, including their DDL (Section 4.2.3), runtime implementation (Section 4.2.4), and lifecycle (Section 4.2.5).

4.2.1 Storage Management in AsterixDB

AsterixDB [30] takes component-based Log-Structured Merge-trees [82] as the storage technology for all of its index structures, including its B+-tree, R-Tree, inverted keyword, and inverted ngram indexes. With LSM-indexes, writes are batched in in-memory components (e.g., in-memory B+-trees) first until they are full. Full in-memory components are flushed to disks and become on-disk components (e.g., on-disk B+-trees). On-disk components are merged at a certain point based on a certain merge policy [30]. AsterixDB provides record-level ACID (atomicity, consistency, isolation, durability) transactions — transactions begin and end implicitly for each inserted, deleted, or searched record when an insert, delete, or query statement is being executed. Transaction read/write locks are only acquired on primary keys for primary index accesses, based on two-phase locking (2PL) [25]. Secondary index accesses are not locked. To guarantee record-level read consistency, secondary index searches are post-validated when accessing records from the primary index. AsterixDB employs a no-steal and no-force update policy according to which in-memory components cannot be flushed until the log records of the corresponding batched modifications are safely on the log disk. Group commits are used to optimize log I/O.

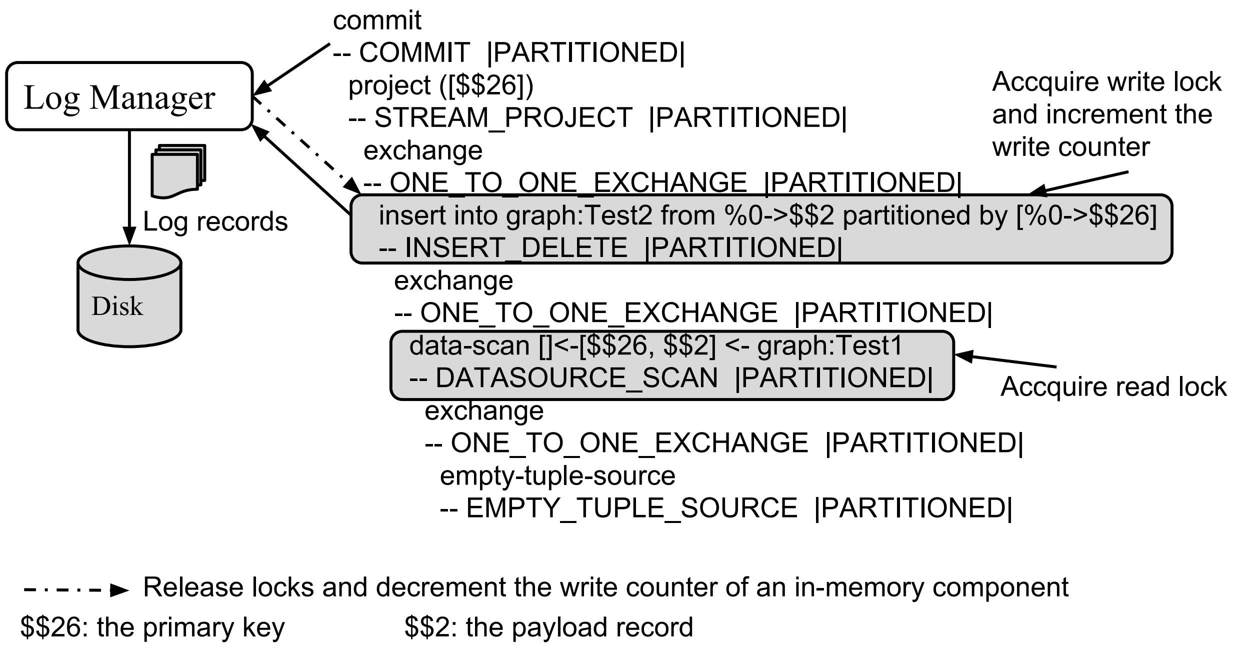
Let us consider two example AQL [29] queries to walk through the mechanisms. The first is an insert query:

```
use dataverse graph;

insert into dataset Test2
(
  for $t in dataset Test1 return $t
)
```

In this query, let us assume that datasets `Test1` and `Test2` have identical record types and identical primary keys. Figure 4.2 shows the query plan. In the resulting AsterixDB query plan, which is best read bottom-up. As annotated in the query plan, a read lock (on a primary key in `Test1`) will be acquired before reading a record from the source dataset `Test1` and a write lock (on the primary key in `Test2`) will be acquired before inserting the record into destination dataset `Test2`. Also, before inserting the record into an in-memory component of dataset `Test2`, a write counter associated with the component will be incremented and a write-ahead log record will be appended to the log buffer of the log manager. In the commit operator, a commit log record for each inserted record's primary key will be appended to the log buffer too. The log manager writes the log records to the log disk in a batch-based asynchronous manner (a.k.a., group commit). Once a commit log record has reached the log disk, the read and write locks on the primary key contained in the commit log record are both released and the write counter in the corresponding in-memory component is decremented. If a state is reached when all in-memory components of dataset `Test2` are full, further inserts will be blocked until one of the in-memory components is flushed to become an on-disk component — and a component can only be flushed when its write counter is zero (i.e., when it contains no un-committed writes).

The second query is a search query:



```
commit
-- COMMIT  |PARTITIONED|
  project ([$$26])
  -- STREAM_PROJECT  |PARTITIONED|
    exchange
    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
      insert into graph:Test2 from %0->$$2 partitioned by [%0->$$26]
      -- INSERT_DELETE  |PARTITIONED|
        exchange
        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
          data-scan []<-[$$26, $$2] <- graph:Test1
          -- DATASOURCE_SCAN  |PARTITIONED|
            exchange
            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
              empty-tuple-source
              -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Log Manager
Log records
Disk
Accquire write lock and increment the write counter
Accquire read lock
Release locks and decrement the write counter of an in-memory component
$$26: the primary key
$$2: the payload record

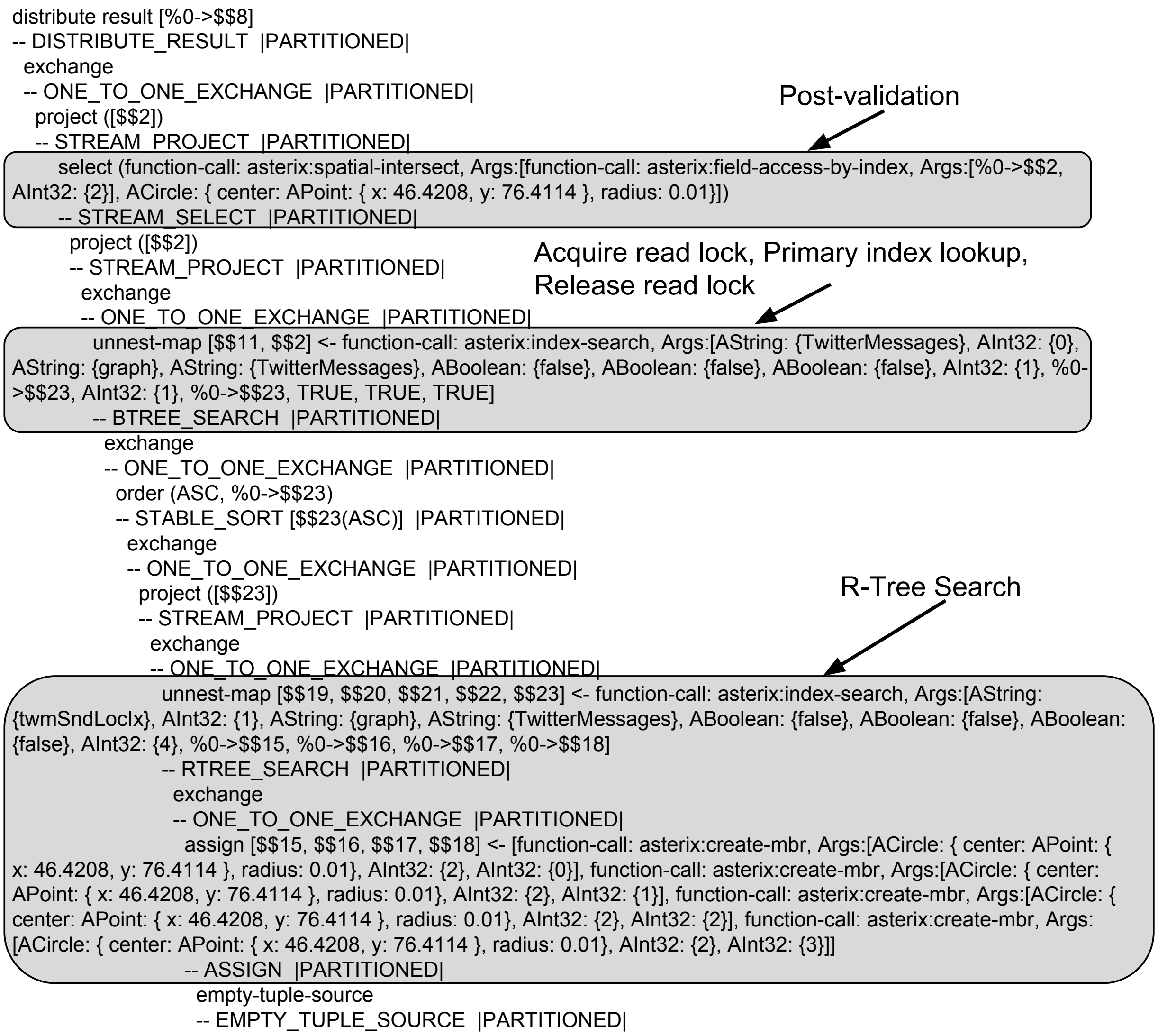
Figure 4.2: The query plan of a scan-insert query.

```
use dataverse graph;

let $p := create-point(46.4208 , 76.4114 )
let $r := create-circle($p, 0.05)

for $tm in dataset TwitterMessages
where spatial-intersect($tm.sender-location , $r)
return $tm;
```

Let us assume that there is a secondary R-Tree index on the field `sender-location` on the dataset `TwitterMessages`. The query plan is shown in Figure 4.3. As we can see from the query plan, the R-Tree search is not locked at all, but each primary index lookup for a fetched primary key from the R-tree is protected by a read lock. Also, for consistency, each retrieved record from the primary index will be checked again against the query predicate in the `Select` operator as a post validation step before it is added as a search result.



```
distribute result [%0->$$8]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$2])
    -- STREAM_PROJECT  |PARTITIONED|
      select (function-call: asterix:spatial-intersect, Args:[function-call: asterix:field-access-by-index, Args:[%0->$$2,
AInt32: {2}], ACircle: { center: APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}])
      -- STREAM_SELECT  |PARTITIONED|
        project ([$$2])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            unnest-map [$$11, $$2] <- function-call: asterix:index-search, Args:[AString: {TwitterMessages}, AInt32: {0},
AString: {graph}, AString: {TwitterMessages}, ABoolean: {false}, ABoolean: {false}, ABoolean: {false}, AInt32: {1}, %0-
>$$23, AInt32: {1}, %0->$$23, TRUE, TRUE, TRUE]
            -- BTREE_SEARCH  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                order (ASC, %0->$$23)
                -- STABLE_SORT [$$23(ASC)]  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    project ([$$23])
                    -- STREAM_PROJECT  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        unnest-map [$$19, $$20, $$21, $$22, $$23] <- function-call: asterix:index-search, Args:[AString:
{twmSndLocIx}, AInt32: {1}, AString: {graph}, AString: {TwitterMessages}, ABoolean: {false}, ABoolean: {false}, ABoolean:
{false}, AInt32: {4}, %0->$$15, %0->$$16, %0->$$17, %0->$$18]
                        -- RTREE_SEARCH  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            assign [$$15, $$16, $$17, $$18] <- [function-call: asterix:create-mbr, Args:[ACircle: { center: APoint: {
x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {0}], function-call: asterix:create-mbr, Args:[ACircle: { center:
APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {1}], function-call: asterix:create-mbr, Args:[ACircle: {
center: APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {2}], function-call: asterix:create-mbr, Args:
[ACircle: { center: APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {3}]]
                            -- ASSIGN  |PARTITIONED|
                              empty-tuple-source
                              -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Figure 4.3: The query plan of a secondary index search query.

In the distributed case, a dataset has its primary and secondary indexes partitioned by the primary key. The metadata of datasets and indexes are recorded in the system's metadata datasets.

4.2.2 Motivation

Record-level ACIDity guarantees that AsterixDB will work correctly in mission-critical environments. However, in real-world data science use cases, data scientists often do experimental and exploratory analysis — temporary, intermediate datasets are created from time to time, to be used repeatedly for visualizations and summarizations or for gluing together different analytical systems. For instance, in the Twitter influence ranking example in Section 4.1, the intermediate tables (datasets) `MsgGraph` and `Result` are used as a data exchange media between a tabular data processing runtime (Hive) and a graph processing runtime (Giraph), and they are only accessed by one data scientist in a serial fashion. The two materialized temporary tables might be accessed again later for other analytical tasks. In this case, ACIDity becomes an overkill that brings in few benefits but will hurt performance, both for this data scientist and for competing the cluster's physical resources. In order to give users an alternative "unsafe" but useful option for the scenario. We have added AsterixDB support for temporary datasets which do not have the same ACID guarantees. In addition, regular datasets are logically "heavy" — including registration in the overall system catalogs, and logically potentially polluting the system's persistent namespace. This is avoided with a new temporary dataset support mechanism.

4.2.3 Temporary Dataset DDL

We add the "temporary" keyword to the create dataset statement of AsterixDB [29]. For instance, similar to the example in Section 4.1, we can create a temporary dataset for the constructed graph from tweets in AsterixDB by executing the following DDL statements:

```
use dataverse graph;

drop type VertexType if exists;
drop dataset MsgGraph if exists.

create type VertexType as{
   vertexid: int64,
   value: double,
   edges: [int64]
};

create temporary dataset MsgGraph(VertexType) primary key vertexid;
```

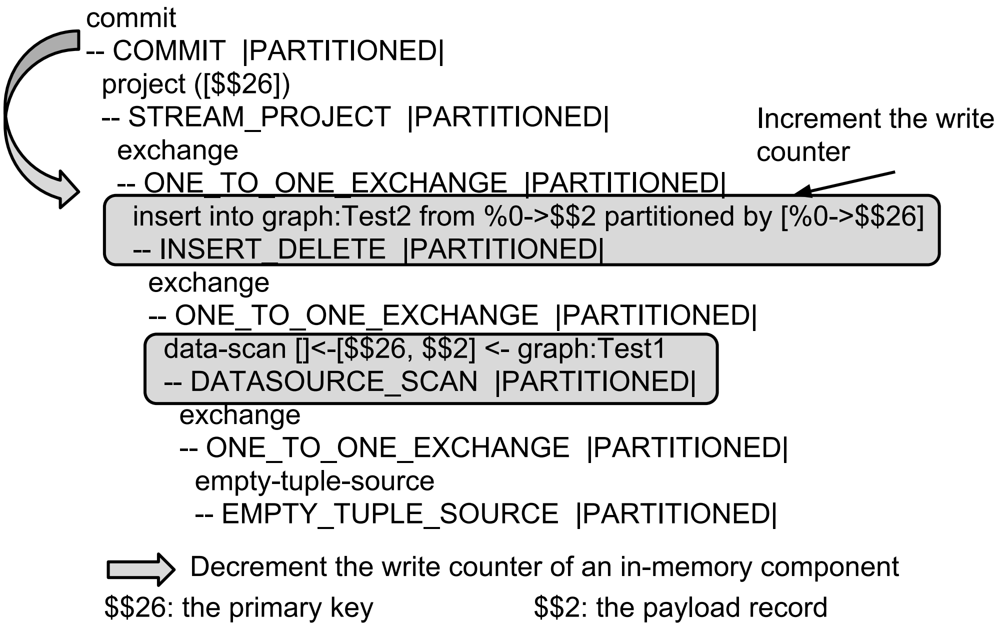
Other DDLs like `create index` and `drop dataset` for temporary datasets are then the same as for regular datasets.

4.2.4 Runtime Implementation

At the logical level, a temporary dataset is different from a regular dataset in three aspects:

- it is not locked, either for reads or writes;
- there is no transaction log for writes, hence there is no restorative crash recovery;
- an indexes (primary or secondary) are not forced to disk after a bulk load (i.e., a load dataset statement, or an internal merge of its on-disk components).

AsterixDB internally has already used a pluggable architecture for locking and logging, where the locking and logging behaviors associated with the index point search operator, index range search operator, and index insert/delete operator can be plugged-in by providing them logging- and locking- related hook interface implementations. Hence, we can use this to add temporary dataset-specific implementations that include no locking nor WAL (write-ahead log) activities. We must also add a branch in the commit operator for temporary



```
commit
-- COMMIT  |PARTITIONED|
  project ([$$26])
  -- STREAM_PROJECT  |PARTITIONED|
    exchange
    -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
      insert into graph:Test2 from %0->$$2 partitioned by [%0->$$26]
      -- INSERT_DELETE  |PARTITIONED|
        exchange
        -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
          data-scan []<-[$$26, $$2] <- graph:Test1
          -- DATASOURCE_SCAN  |PARTITIONED|
            exchange
            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
              empty-tuple-source
              -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Increment the write counter
Decrement the write counter of an in-memory component
$$26: the primary key
$$2: the payload record

Figure 4.4: The query plan of a scan-insert query for temporary datasets.

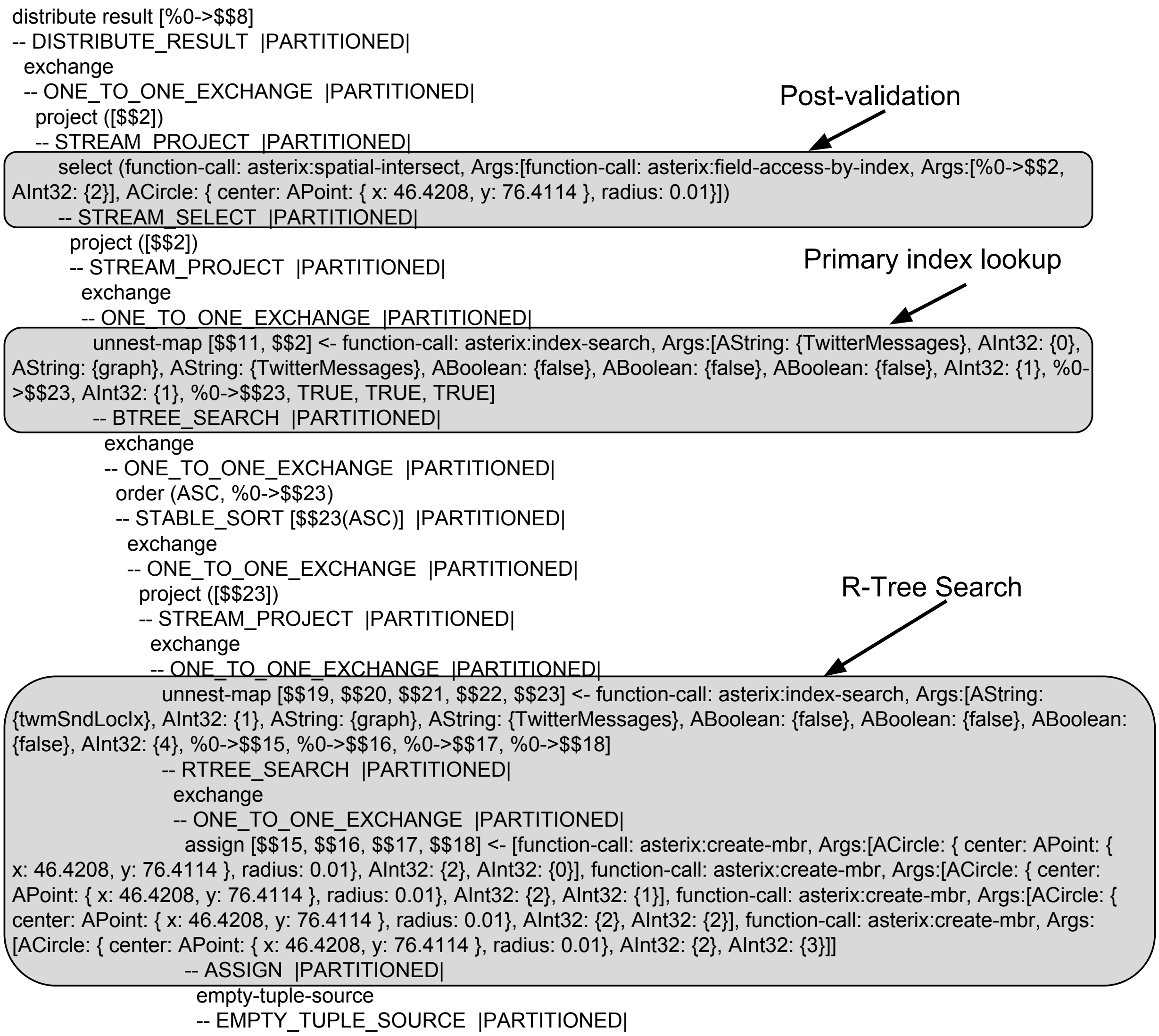
datasets, one where instead of appending a commit log record to the log buffer of the log manager, it just decrements the writer count of the in-memory component that stores the insert (or delete).

If the datasets `Test1` and `Test2` in the example of Section 4.1 are replaced with temporary datasets, the query plans of the scan-insert query and the secondary index search query become the ones in Figures 4.4 and 4.5 respectively. (Note that in Figure 4.5, a post-validation step is still necessary to guarantee that the returned results of the R-tree search query do not contain false positives.)

While locks are avoided, latches are still required for temporary datasets in the current implementation, as temporary datasets share the same disk buffer cache with regular datasets.

4.2.5 Temporary Dataset Lifecyle

A temporary dataset has the following lifecyle properties:



```
distribute result [%0->$$8]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$2])
    -- STREAM_PROJECT  |PARTITIONED|
      select (function-call: asterix:spatial-intersect, Args:[function-call: asterix:field-access-by-index, Args:[%0->$$2,
AInt32: {2}], ACircle: { center: APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}])
      -- STREAM_SELECT  |PARTITIONED|
        project ([$$2])
        -- STREAM_PROJECT  |PARTITIONED|
          exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
            unnest-map [$$11, $$2] <- function-call: asterix:index-search, Args:[AString: {TwitterMessages}, AInt32: {0},
AString: {graph}, AString: {TwitterMessages}, ABoolean: {false}, ABoolean: {false}, ABoolean: {false}, AInt32: {1}, %0-
>$$23, AInt32: {1}, %0->$$23, TRUE, TRUE, TRUE]
            -- BTREE_SEARCH  |PARTITIONED|
              exchange
              -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                order (ASC, %0->$$23)
                -- STABLE_SORT [$$23(ASC)]  |PARTITIONED|
                  exchange
                  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                    project ([$$23])
                    -- STREAM_PROJECT  |PARTITIONED|
                      exchange
                      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                        unnest-map [$$19, $$20, $$21, $$22, $$23] <- function-call: asterix:index-search, Args:[AString:
{twmSndLocIx}, AInt32: {1}, AString: {graph}, AString: {TwitterMessages}, ABoolean: {false}, ABoolean: {false}, ABoolean:
{false}, AInt32: {4}, %0->$$15, %0->$$16, %0->$$17, %0->$$18]
                        -- RTREE_SEARCH  |PARTITIONED|
                          exchange
                          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                            assign [$$15, $$16, $$17, $$18] <- [function-call: asterix:create-mbr, Args:[ACircle: { center: APoint: {
x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {0}], function-call: asterix:create-mbr, Args:[ACircle: { center:
APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {1}], function-call: asterix:create-mbr, Args:[ACircle: {
center: APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {2}], function-call: asterix:create-mbr, Args:
[ACircle: { center: APoint: { x: 46.4208, y: 76.4114 }, radius: 0.01}, AInt32: {2}, AInt32: {3}]]
                            -- ASSIGN  |PARTITIONED|
                              empty-tuple-source
                              -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

Figure 4.5: The query plan of a secondary index look on a temporary dataset.

- It cannot sustain an AsterixDB instance restart — each slave node clears the directory for temporary datasets at node startup or restart time;
- It will be garbage-collected if it has not been accessed in the last 30 (or other chosen interval) days — a background garbage collector thread is in charge of that. (A data scientist may need to create a large number of temporary datasets and may forget to drop some of them from time to time.)

- Its metadata (including the metadata of all its secondary indexes) is not stored in the AsterixDB metadata store [29]. Instead, it is only kept in the master machine's memory, so an instance restart will lose the metadata.

In summary, temporary datasets offer an AsterixDB user a more lightweight way of generating, storing, and scanning temporary data as are supported, but still provide support for more flexible access patterns for the temporary data, e.g., full queries including secondary index searches are supported. We will investigate their performance benefits in Section 4.5.

4.3 The External Connector Framework

The temporary datasets proposed in Section 4.2 offer an "unsafe" but lightweight option for storing data in AsterixDB for data science use cases. This section explores the other side of rich(er) graph analytics — how to build an extensible architecture for a special-purpose Big Data analytics engine like Pregelix to process data from various data sources, e.g., HDFS, AsterixDB, and many others. Section 4.3.1 explains the motivation, Section 4.3.2 introduces a generalized connector framework for Pregelix, and Section 4.3.3 describes the implementation of an AsterixDB connector for Pregelix on top of the generalized connector framework.

4.3.1 Motivation

Nowadays, there are more and more available storage options for enterprise data lakes. For example, distributed file systems like HDFS [10] are typically used for storing archived raw data, e.g., machine logs, click streams, and browse histories; NoSQL stores like HBase [12], Cassandra [2], MongoDB [17], and Couchbase [3] are usually used to support fast point reads and writes for online applications, such as mobile games, transportation apps, and e-commerce websites; distributed search engines like Elastic Search [5] are often used for storing and searching textual data; AsterixDB is also an option for ingesting and more richly querying semi-structured data. Each storage option fits a particular category of applications.

The Pregelix system we described in Chapter 3 only supports HDFS as its data source and sink. A naïve way to allow Preglix to interact with different storage systems is to simply adopt HDFS as the glue media, similar to the example in Section 4.1. However, that approach has two drawbacks:

- it adds extra HDFS read-write round trips;
- simply using the client query API (e.g., REST API [58]) offered by a storage system may limit the parallelism of loading data into Pregelix.

These shortcomings have motivated us to build an extensible connector framework (Section 4.3.2) for Pregelix to directly read from and write to external storage systems. To demonstrate the usefulness of the framework and allow AsterixDB users to enjoy graph analytics capabilities, we have used it to implement an AsterixDB connector for Pregelix (Section 4.3.3).

4.3.2 The Connector Framework

To support various data sources and sinks for Pregelix, we abstract its data readers, writers, and data transformers as `parallel operator`s. A parallel operator encapsulates two aspects:

- what actual work should be done at a single parallel partition level;
- where its actual worker threads should be run.

Besides parallel operators, we encapsulate how the external data is partitioned and sorted as `physical properties`. Physical properties can be used to generate partitioners and sorters at runtime for each partition of the corresponding external data.

For the `parallel operator` abstraction, the read connector interface is as follows:

```
public interface IReadConnector {
    /** Returns the data read parallel operator. */
    public ParallelOperator getReadOperatorDescriptor();

    /** Returns the data transformation(from "external" to Pregelix)  parallel operator. */
    public ParallelOperator getReadTransformOperatorDescriptor();
}
```

Similarly, the write connector interface is as follows. In addition to the write and transform parallel operators, a write connector should also encapsulate information about physical properties required by the sink storage system.

```
public interface IWriteConnector {
    /** Returns the physical properties of the data sink. */
    public PhysicalProperties getPhysicalProperties();

    /** Returns the data write parallel operator. */
    public ParallelOperator getWriteOperatorDescriptor();

    /** Returns the data transformation (from Pregelix to "external") parallel operator. */
    public ParallelOperator getWriteTransformOperatorDescriptor();
}
```

Figure 4.6 demonstrates how the generated runtime operators of read and write connectors are articulated with a Pregelix job. In the figure, readers 1-3 and transformers 11-13 are generated from a read connector, while transformers 21-23 and writers 1-3 are generated from a write connector. How data is redistributed from transformers 11-13 to Pregelix partitions is dictated by the Pregelix computation's partitioning requirement. Differently, how data

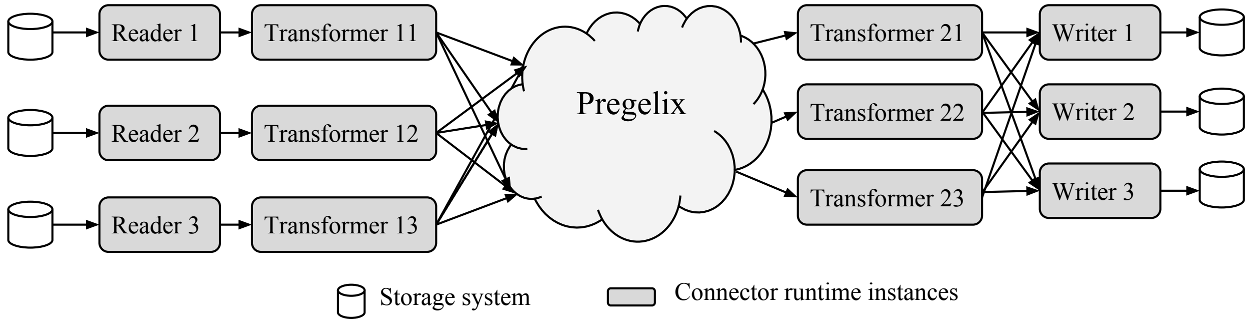


Figure 4.6: The connector runtime dataflow.

is redistributed from transformers 21-23 to writers 1-3 are determined by the sink storage system's requirement, i.e., by the physical properties of the write connector. Note that sorting operators may need to be injected before the writers to sort data locally, depending on the requirement of the destination storage system, which is also contained as part of the physical properties of the write connector. The connector framework is flexible enough to generate different dataflows based on the physical properties of different write connectors.

In the example of Figure 4.6, inside the Pregelix job, the redistributed output of transformers 11-13 are sorted and bulk-loaded into partitioned Pregelix vertex B-trees (as described in Section 3.5.2); the inputs of transformers 21-23 come from scanning partitioned vertex B-Trees in Pregelix. Other than the data loading and result writing parts, the main Pregelix computation dataflows (i.e., supersteps) remain the same as those described in Chapter 3.

4.3.3 The AsterixDB Connector

For the purpose of implementing the read and write connectors of AsterixDB, knowledge of where an AsterixDB dataset is partitioned and stored, as well as how the binary representation of a record in the dataset should be parsed (or constructed), should be able to be obtained from outside of AsterixDB. Thus, we have added a REST API [58] in AsterixDB to enable outside connector implementations to retrieve this meta information for a dataset. The API's input includes a `dataverse` name (i.e., the AQL equivalent of a `database` in SQL [20]) and a `dataset` name (i.e., the AQL equivalent of `table` in SQL). The API's output is a JSON [15] object with four fields:

- record type, the type description of records in the dataset;
- primary key, the name(s) of the primary key field(s) of the dataset;
- temporary dataset flag, indicating whether the dataset is temporary or not;
- partitioned files, a list of file descriptors with each containing an absolute file path and the IP address of its hosting machine.

Using this new API, our AsterixDB/Pregelix read and write connector implementations pull the meta information of a dataset from the REST API and then construct parallel operators and physical properties based on the retrieved information. With the partitioned files, the read (or write) connector is able to create LSM B+-tree scanners (or bulk-loaders) to read (or write) those files. With the record type, the read (or write) connector can create read (or write) transformation operators that parse (or build) AsterixDB records correctly. With the primary key, the write connector can create its physical properties, which can then generate partitioners and sorters for bulk loading the target AsterixDB dataset. In our implementation, both the read and write connectors check the temporary dataset flag and error out accesses on regular datasets; this protects the ACIDity of regular datasets.

In addition to this new metadata API, AsterixDB exposes another REST API to allow outsiders to request to a synchronous flush of the in-memory components of a dataset to the OS file system. The read connector issues such a request before the Pregelix data loading runtime starts to make sure that the latest, in-memory batched updates of the dataset can be read.

From a Pregelix user's perspective, in order to read an input graph from AsterixDB, s/he needs to provide a VertexInputConverter interface implementation (as follows). Similarly, in order to write an output graph states to AsterixDB, s/he needs to provide a VertexOutputConverter interface implementation.

```java
public interface VertexInputConverter {
    /** A user-defined function to covert an AsterixDB record to a vertex. */
    public void convert(ARecordVisitablePointable recordPointable, Vertex vertex);
}

public interface VertexOutputConverter {
    /** A user-defined function to covert a vertex to an AsterixDB record. */
    public void convert(Vertex vertex, RecordBuilder recordBuilder);
}
```

The `convert` methods in the two interfaces above are called from read and write transformation operators respectively. The two operators follow the design paradigm that we propose in Chapter 2 to avoid unnecessary Java object creations. A user can implement arbitrary conversions, however, so s/he must guarantee that both the resulting vertex ids and the resulting primary key values are globally unique.

In Pregelix, we provide a default implementation of the two converters for user convenience. The default VertexInputConverter can parse a AsterixDB record of the form (vertexid, value, edgelist) into a vertex, and the default VertexOutputConverter can build a record of the form (vertexid, value) from an input vertex. Users with more sophisticated needs can then replace these with their own customized implementations.

With the AsterixDB connector, a Pregelix job can take either an AsterixDB temporary dataset or a HDFS path as the input data source, and it can take either of them as the output data sink. If an AsterixDB temporary dataset is taken as the data sink, any preexisting records in the dataset will be erased as a result of executing the new Pregelix job.

4.4 Running Pregel from AsterixDB

With the infrastructures built in Section 4.2 and Section 4.3, we have added a `run pregel` statement into the DML of AQL [29]. The syntax of the statement is as follows:

```
run pregel ( <jar-path>,  <full-class-name>, <args>)
from dataset <dataset-name>
to dataset <dataset-name>
```

This new statement is compiled into a Pregelix job that takes the dataset after `from` as the data source and the dataset after `to` as the data sink. The jar path specifies the location of the job's binary, the full class name points to a user-defined job client entrance class, and the argument string is for the main method in the job client entrance class.

Let us revisit the example in Section 4.1. To achieve the same analytical task in AQL, we first run several DDLs to create a `TwitterMessages` dataset as well as the data types for two temporary intermediate datasets as follows.

```
use dataverse graph;

// Creates the Twitter message type.
create type TwitterMessageType as {
   tweetid: int64,
   user: string,
   sender-location: point,
   send-time: datetime,
   reply-from: int64,
   retweet-from: int64,
   referred-topics: {{ string }},
   message-text: string
}

// Creates the TwitterMessages dataset.
create dataset TwitterMessages(TwitterMessageType) primary key tweetid;

// Loads the TwitterMessages dataset.
......

// Creates the data types for the intermediate temporary datasets.
create type VertexType as{vertexid: int64, value: double, edges: [int64]};
create type Result as {vertexid: int64,  rank: double};
```

Having the `run pregel` statement, we can now implement the example in Section 4.1 as follows in AQL (`GQ1`).

GQ1: Twitter Impact Ranking

```
use dataverse graph;

// DDLs that cleans up and creates temporary datasets.
drop dataset MsgGraph if exists;
drop dataset results if exists;
create temporary dataset MsgGraph(VertexType) primary key vertexid;
create temporary dataset results(Result) primary key vertexid;

// Populate temporary dataset MsgGraph.
insert into dataset MsgGraph
(
   for $tm in dataset TwitterMessages
   let $edge:=switch-case($tm.reply-from>=0, true, $tm.reply-from, false, $tm.retweet-from)
   where $edge>=0
   return {  "vertexid": $tm.tweetid, "value": 1.0/10000000000.0, "edges": [$edge] }
)

// Runs a Pregelix job. The job uses default input/output converters.
run pregel("examples/pregelix-example-jar-with-dependencies.jar",
 "edu.uci.ics.pregelix.example.PageRankVertex",  "-num-iteration 5")
from dataset MsgGraph
to dataset results;

// Final query that ranks users.
for $tm in dataset TwitterMessages
for $r in dataset results
where $r.vertexid = $tm.tweetid
group by $un := $tm.user with $r
let $rank := sum(for $i in $r return $i.rank)
order by $rank desc limit 50
return {  "name": $un,  "rank": $rank };
```

In this example, the logic of both querying tabular data and running graph algorithms from a Pregelix library are written in AQL. A data scientist does not know how datasets like `MsgGraph` and `results` are physically stored, no longer need to think about gluing different systems through HDFS, and can just focus on writing the right queries and implementing and then employing the right "think-as-a-vertex" graph algorithms. (Speedup and scale-up properties of GQ1 will be experimentally evaluated in Section 4.5.3.)

In addition to the improved user experience, the secondary index support provided by AsterixDB could make the following graph query use cases (GQ2) run at an interactive speed (i.e. within-a-minute response time) if there is a R-Tree secondary index on the sender-location field of tweets.

GQ2: Twitter Impact Ranking in a Given Spatial Region

```
use dataverse graph;

// DDLs that cleans up and creates temporary datasets.
drop dataset MsgGraph if exists;
drop dataset results if exists;
create temporary dataset MsgGraph(VertexType) primary key vertexid;
create temporary dataset results(Result) primary key vertexid;

// Populates temporary dataset MsgGraph.
insert into dataset MsgGraph
(
   let $p := create-point(46.4208 , 76.4114 )
   let $r := create-circle($p, 0.05)

   for $tm in dataset TwitterMessages
   let $edge:=switch-case($tm.reply-from>=0, true, $tm.reply-from, false, $tm.retweet-from)
   where $edge >= 0 and spatial-intersect($tm.sender-location, $r)
   return { "vertexid": $tm.tweetid, "value": 1.0/10000000000.0,  "edges": [$edge] }
);

// The Pregel job and the final query are the same as that in GQ1.
......
```

Since we know upfront that dataset results would not be very large and that field tweetid is the primary key for the TwitterMessages dataset, the indexnl hint in the last ranking query lets the AsterixDB query optimizer generate an index nested loop join which avoids redistributing the entire `TwitterMessages` dataset for a hash join. The combination of the indexing capability and the Pregel abstraction makes interactive graph analytics possible. (Performance results will be reported in Section 4.5.4.)

Last but not least, there is another category of applications where the graph extraction query can be more complex than that in `GQ1` and `GQ2`. The following query (`GQ3`) is such an example, which is to analyze and rank people's instant influences on Twitter. This query only extracts replies and retweets that are posted within a day from the time when their referred messages are posted. With the join capabilities as well as temporal data types in AsterixDB, this query could also be simply expressed and efficiently evaluated. (Experimental results will be discussed in Section 4.5.5.)

GQ3: Twitter Instant Impact Ranking

```
use dataverse graph;

// DDLs that cleans up and creates temporary datasets.
drop dataset MsgGraph if exists;
drop dataset results if exists;
create temporary dataset MsgGraph(VertexType) primary key vertexid;
create temporary dataset results(Result) primary key vertexid;

// Populates temporary dataset MsgGraph.
insert into dataset MsgGraph
(
   for $tm2 in dataset TwitterMessages
   for $tm1 in dataset TwitterMessages
   let $edge:=switch-case($tm1.reply-from>=0,true,$tm1.reply-from,false,$tm1.retweet-from)
   let $d := duration-from-interval(interval-from-datetime($tm1.send-time, $tm2.send-time))
   where $edge>=0 and $edge=$tm2.tweetid and $d<= day-time-duration("P1D")
   return { "vertexid": $tm1.tweetid, "value": 1.0/10000000000, "edges": [$tm2.tweetid] }
);

// The Pregel job and the final query are the same as that in GQ1.
......
```

Name	Size	Number of Records
1×	440 GB	1,711,777,760
0.5×	220 GB	855,829,620

Table 4.1: The tweet dataset and its scale-downs.

4.5 Experiments

This section reports an initial experimental evaluation of the techniques developed in this chapter. Section 4.5.1 describes the experimental settings; Section 4.5.2 compares temporary datasets and regular datasets with respect to write performance; Section 4.5.3, Section 4.5.4, and Section 4.5.5 report the scaling properties of a long-running graph analytical query (GQ1, as described in Section 4.4), an interactive graph query (GQ2, as described in Section 4.4), and a graph query with a complex ETL (GQ3, as described in Section 4.4) respectively; finally, Section 4.5.6 summarizes the numbers.

4.5.1 Experimental Setup

We ran the experiments detailed here on a 16-node Linux IBM x3650 cluster with one additional master machine of the same configuration. Nodes are connected with a Gigabit Ethernet switch. Each node has one Intel Xeon processor E5520 2.26GHz with four cores, 8GB of RAM, and two 1TB, 7.2K RPM hard disks.

In the experiments, we used a synthetic tweet dataset in JSON format of the record type `TwitterMessageType` as described in Section 4.4. In the dataset, 10% of the tweets are retweets and another 10% are replies. If a tweet is an original message, the `reply-from` field and the `retweet-from` field are both -1. Table 4.1 shows size statistics for the synthetic datasets, including the full dataset (1×) as well as a scale-down sample (0.5×).

Parameter	Value
Memory budget (for LSM in-memory components) per dataset	256 MB
Data page size	128 KB
Disk buffer cache size	1GB
Bloom filter target false positive rate	1%
Log buffer size (for regular datasets)	4MB
Number of data partitions	4
JVM maximum heap size	6GB

Table 4.2: AsterixDB per-machine settings used throughout these experiments.

For all the experiments in this section, we used the AsterixDB per-machine settings listed in Table 4.2. Before running the evaluation queries, we loaded the generated tweets into the dataset `TwitterMessages` (as described in Section 4.4) and created an R-Tree secondary index on the `sender-location` field for the dataset.

4.5.2 Evaluations of Temporary Datasets

In this experiment, we ran the following insert query on a 16-machine AsterixDB instance with the 1× dataset:

```
insert into dataset MsgGraph2{
  for $m in dataset MsgGraph
  return $m
}
```

In the query, dataset `MsgGraph` is a temporary dataset that resulted from the insert query in GQ1 (described in Section 4.4) — it contains 342,366,948 records and has 23.36 GB. Both `MsgGraph` and `MsgGraph2` have identical record types and primary keys. Before the insert, dataset `MsgGraph2` is empty. We compared the two cases when dataset `MsgGraph2` is a regular dataset versus when it is a temporary dataset. We also compared the two cases when the semantics of insert mean "insert if not exists, else raise an error if the key exists" (the default

Metric	Temporary Dataset	Regular Dataset
Insert Time (ms)	256,224	712,552
Insert TPS (per partition)	20,876	7,507
Insert TPS (on cluster)	1,336,097	480,479
Upsert Time (ms)	127,068	642,369
Upsert TPS (per partition)	42,099	8,327
Upsert TPS (on cluster)	2,694,360	532,953

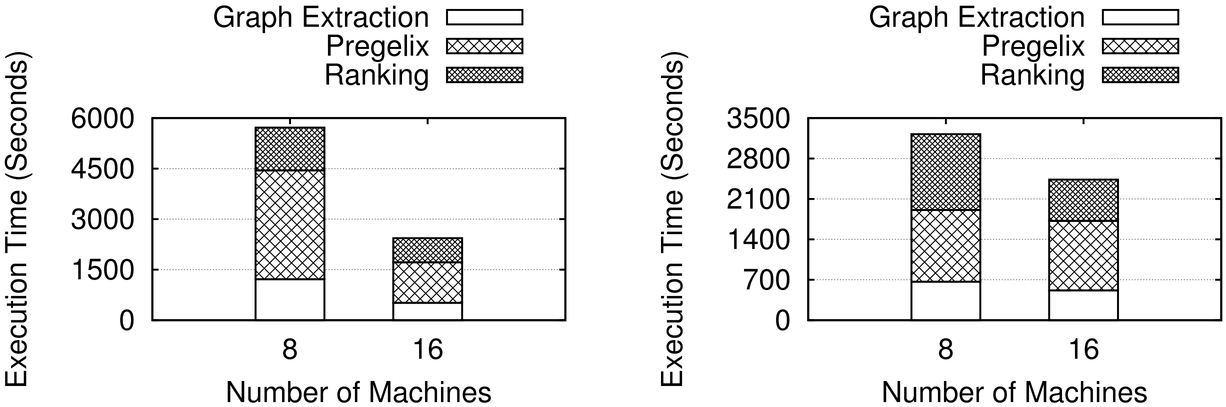
Table 4.3: Insert/Upsert performance: temporary datasets V.s. regular datasets.

AsterixDB semantics) versus when the semantics of insert mean "insert if not exists, else update" (via a private change of a flag in the AsterixDB source code). We will use the term `insert` to represent the former semantics and use the term `upsert` to represent the latter semantics. We evaluated the `upsert` option in addition to the default `insert` option because we wanted to understand relative overhead of logging and searching (for duplicates) and `upsert` delivers the same semantics for this particular query as `insert` (it is guaranteed to have no duplicates in the input stream and a temporary dataset is only used in a serial manner). Table 4.3 shows all of the results.

As we can see from Table 4.3, temporary datasets obtain a 2.78× speedup here for `insert` and a 5.05× speedup for `upsert`; these are because of the reduced logging and locking overheads. Interestingly, going from `insert` to `upsert`, regular datasets do not enjoy significant speedups. One reason is that in `upsert`, though the search overhead is removed, the system is not bound by CPU time but by the I/Os for forcing the transaction log. The other reason is that in our experiments, we did not use dedicated log disk devices, which consequently caused much more random I/Os.

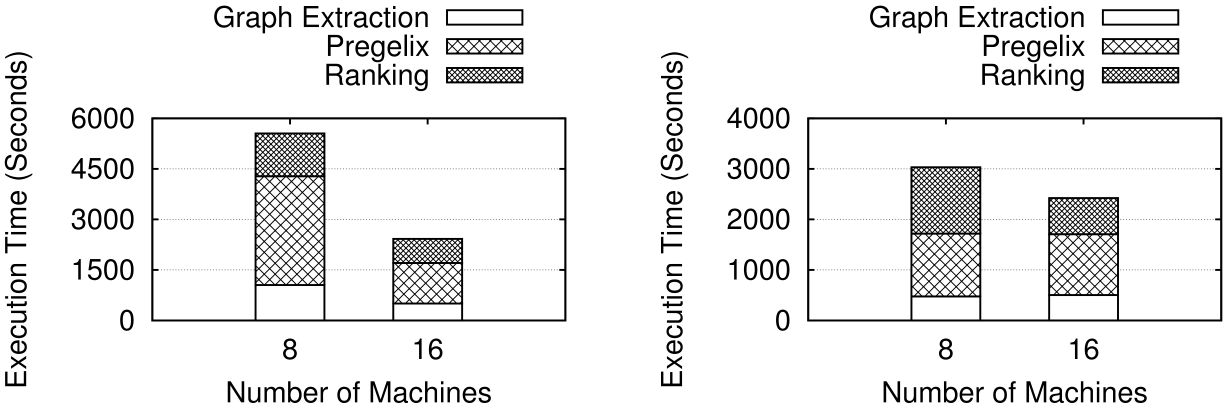
4.5.3 Evaluation of GQ1 (Long-Running Analytics)

In this experiment, we tested the parallel speedup and scale-up of GQ1 (described in Section 4.4) using two AsterixDB instances — one had 16 slave machines and the other had 8



(a) Parallel speedup (b) Parallel scale-up

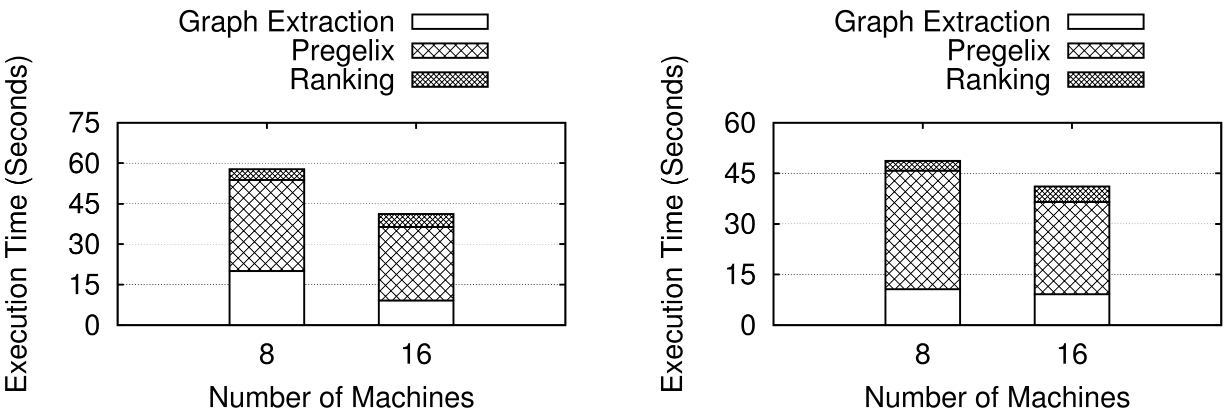
Figure 4.7: The parallel speedup and scale-up of GQ1 (insert).



(a) Parallel speedup (b) Parallel scale-up

Figure 4.8: The parallel speedup and scale-up of GQ1 (upsert).

slave machines. In the speedup experiment, we ran GQ1 using the 1× dataset on the two AsterixDB instances. In the scale-up experiment, GQ1 was run with the 1× dataset on the 16-machine AsterixDB instance and with the 0.5× dataset on the 8-machine AsterixDB instance. We report numbers with both insert and upsert for the graph extraction AQL insert query. Figure 4.7(a) shows the parallel speedup and Figure 4.7(b) plots the parallel scale-up, both with the insert semantics. Figure 4.8(a) and Figure 4.8(b) are the corresponding results for the upsert semantics. The numbers show that the parallel speedup and scale-up of `GQ1` is very good.



(a) Parallel speedup

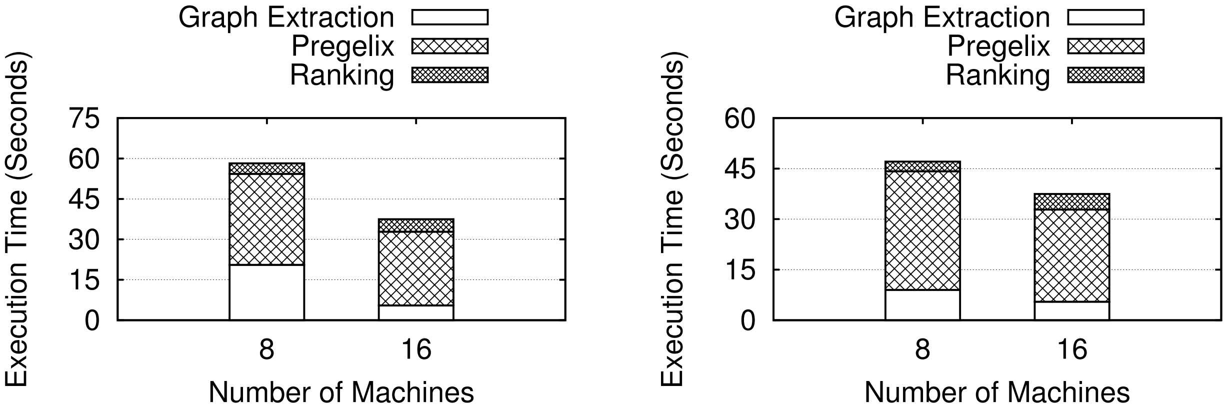
(b) Parallel scale-up

Figure 4.9: The parallel speedup and scale-up of GQ2 (insert).

4.5.4 Evaluation of GQ2 (Interactive Analytics)

In this experiment, we ran GQ2 (described in Section 4.4)) using the same settings as Section 4.5.3. The `spatial-intersect` filter predicate in GQ2 selects 3,354 matched tweets, and the Pregel computation results contain 6,708 vertices. With the insert semantics, the parallel speedup and scale-up of the query are shown in Figures 4.9(a) and 4.9(b) respectively. Figures 4.10(a) and 4.10(b) report the numbers for the upsert semantics. As we can see from the results, with the help of the indexing capabilities in AsterixDB, queries like GQ2 can be answered much more quickly by avoiding scanning the entire `TwitterMessages` dataset.

While GQ2 is faster than GQ2, in our current implementation the `run pregel` statement is compiled to a shell command that kicks off a Pregelix job; this adds a constant client JVM startup time (taking from 10 to 20 seconds) into the picture. In future work, we would like to avoid this overhead by integrating a native Pregelix job client into AsterixDB. Additionally, Pregelix treats the jar file in a submitted job as a new jar, deploys it to the whole cluster when the job starts, and then erases it when the job finishes. Jar deployment adds another 5 to 10 seconds for each job. We can potentially reduce this cost by allowing a Pregelix job to reuse old deployments for cases when a given graph computation is used multiple times.



(a) Parallel speedup

(b) Parallel scale-up

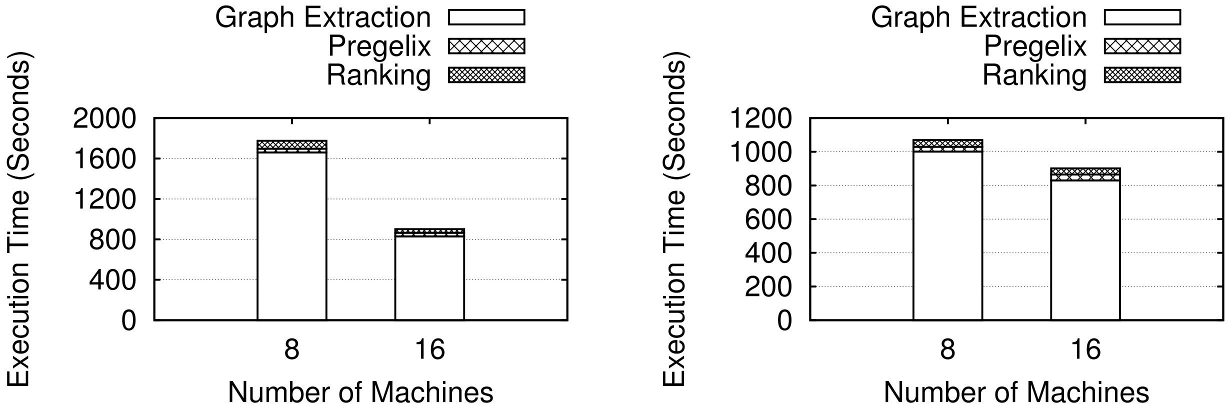
Figure 4.10: The parallel speedup and scale-up of GQ2 (upsert).

4.5.5 Evaluation of GQ3 (Complex Graph ETL)

In this experiment, we ran `GQ3` (described in Section 4.4) using the same settings as Section 4.5.3. `GQ3` inserts 65,353 matched tweets into temporary dataset `MsgGraph`, and the Pregel computation results contain 130,702 vertices. With the insert semantics, the parallel speedup and scale-up of the query are shown in Figures 4.11(a) and 4.11(b) respectively. Figures 4.12(a) and 4.12(b) report the numbers for the upsert semantics. The numbers show very good speedup and scale-up properties of `GQ3`. The reason is that the `insert` statement in `GQ3` dominates the execution time and AsterixDB supports parallel joins well [29] — which is the bulk of the query inside the `insert` statement.

4.5.6 Discussions

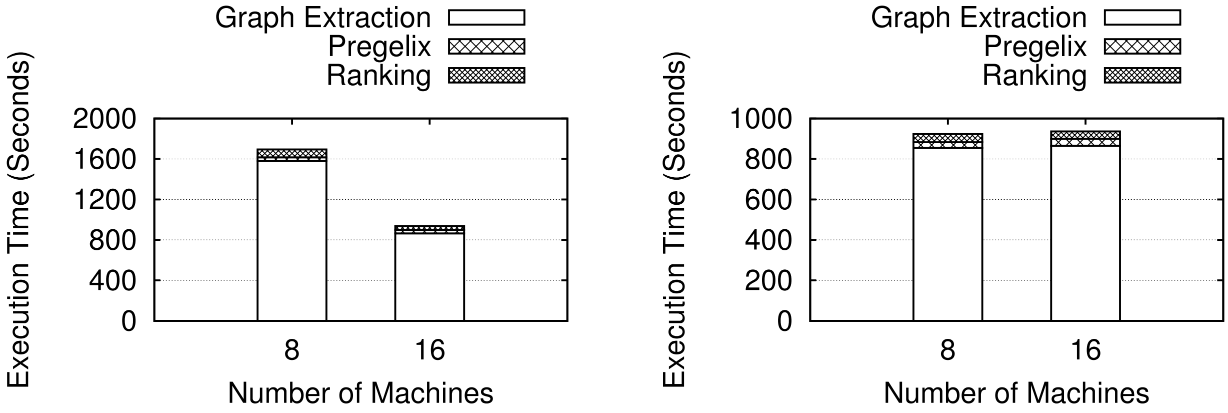
From the numbers reported for `GQ1`, `GQ2`, and `GQ3`, we can see that the combination of AsterixDB and Pregelix can fit a wide-range of use cases. We also find that in these queries, `insert` and `upsert` do not make significant difference for the end-to-end query response time, though `upsert` is shown to be 2× faster than `insert` for temporary datasets in Table 4.3.



(a) Parallel speedup

(b) Parallel scale-up

Figure 4.11: The parallel speedup and scale-up of GQ3 (insert).



(a) Parallel speedup

(b) Parallel scale-up

Figure 4.12: The parallel speedup and scale-up of GQ3 (upsert).

The reason is that in each of the three queries, first, the performance of the `insert` statement is not dominated by the actual inserts or upserts but by other parts in the physical query plan, e.g., scanning, filtering, searching, or joining; second, the end-to-end query response time is not always dominated by the insert statement.

4.6 Related Work

Real-world, large-scale, complex graph analytics have led to significant interest, both in industry and in academia to build efficient and scalable graph processing platforms that can tightly interact with tabular data processing systems. There are extensions from data-parallel platforms to support graph processing as well as extensions from graph computing platforms to support tabular data processing. On one side, GraphX [97], which is built on top of the data-parallel platform Spark [108], provides a programming abstraction called Resident Distributed Graphs (RDGs) to simplify graph loading, construction, transformation, and computations. The integration with Spark allows a user to run a GraphX program together with a wide range of Spark ecosystem tools, such as SparkSQL [35], DataFrames [33], Spark-R [32] and so on. Similarly, in Aster Data, Pregel-like, user-defined graph analytics functions could be invoked from SQL queries [91]. On the other side, distributed GraphLab [73] is adding more and more tabular data processing capabilities into its runtime [87], and Giraph [9] has built connectors for Hive [92] and HBase [12]. However, those latter two graph processing systems still require users to interact with multiple data processing platforms to achieve a complex analytical task, while the GraphX/Spark ecosystem offers users a single, unified entry at the logical level. This chapter has worked towards a similar convergence too, but starting from the Big Data management system (BDMS) side. Similar to GraphX and Aster Data, the integration of Pregelix (Chapter 2) and AQL [29] (as described in Section 4.4) also offers users a single, logical entry for rich(er) graph analytics. However, the indexing capability of AsterixDB, as well its built-in support for semi-structured data and its ability to work against data that does not fit into main memory, further expends the landscape of well-supported analytics — GQ2 is such an example.

4.7 Summary

Complementary to Chapter 3 that investigated how to build a scalable Big Graph analytics platform, the chapter has studied the problem that "graph analytics are not only about graphs" — i.e., a complex graph analytical pipeline often needs to extract graphs, execute distributed graph algorithms, and then run further post-processing queries. We first added an "unsafe" but lightweight storage option, temporary datasets, into AsterixDB, for data science use cases. We then built an external connector framework for Pregelix to access various data sources and we implemented an AsterixDB connector on top of it. Finally, we extended AQL, the query language of AsterixDB, to have a `run pregel` statement. Those techniques combine to offer users a simple, unified way to run rich(er) graph analytical queries. Preliminary experimental evaluations demonstrated that the resulting system has good speedup and scale-up properties for both long-running, offline graph queries and fast, interactive graph queries.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In this thesis, we have studied several issues and challenges in building a scalable software infrastructure for end-to-end Big Graph analytics.

We first investigated a foundational issue for building Big Data platforms — how to write efficient and scalable programs in managed languages. The bloat-aware design paradigm that we described in Chapter 2 is used throughout the entire system stack that we have been building, e.g., Hyracks [42], Algebricks [41], Pregelix (Chapter 3), AsterixDB [29], VXQuery [27], and IMRU [85]. By following this design paradigm, these systems all incur a very limited amount garbage collection overhead and use their memory resources in an economical fashion.

We next explored a new architectural approach to building a Big Graph analytics platform in Chapter 3, where we implemented the Pregel message-passing semantics as iterative dataflows but kept its user-friendly "think-like-a-vertex" client programming abstraction unchanged. Based on this architecture, we built a new open source Big Graph analytics platform called Pregelix. In contrast to other current open source Big Graph analytics platforms, the dataflow-based architecture and the Hyracks data-parallel runtime together give Pregelix the ability to work gracefully with any given level of resources, ranging from a single memory-constrained machine to a large cluster.

Last but not least, we designed and built an integration of Pregelix and AsterixDB in Chapter 4; it provides users a simple, unified logical entry point for richer forms of graph analytics with less ETL pain. With the resulting system, data scientists can work at the logical level of analytical problems without worrying about the physical details of gluing together tabular data processing platforms and graph processing platforms. This exploration also lays a foundation for AsterixDB to incorporate more specialized analytics runtimes to enrich its range of supported analytics.

Based on the design paradigm, architecture, and methodology that we have proposed and evaluated in this thesis, the resulting open-source systems Pregelix (http://pregelix.ics.uci.edu) and AsterixDB (htpp://asterixdb.ics.uci.edu) are available for download and use in support of efficient, scalable, robust, and simple end-to-end Big Graph Analytics.

5.2 Future Work

Further along the path of this thesis, we believe there are several exciting and challenging directions to be explored:

- Currently, the Pregelix system described in Chapter 3 directly generates Hyracks runtime dataflows. It would be interesting to re-architect the system to be based on Algebricks [41], a data model agnostic algebra layer that is now available on top of the Hyracks runtime. With a recursion extension [44], Algebricks should be able to serve as a substrate for

Pregelix. Building the "think-like-a-vertex" programming abstraction on top of a logical algebra layer will further increase the system's physical flexibility, and once Algebricks has a cost-based optimizer, Pregelix can enjoy the benefits as well. Moreover, such an architecture would allow a even tighter and more natural integration with AsterixDB than what Chapter 4 describes; the Pregel semantics could be captured as a function in the AQL query language at the logical level, without requiring materialized temporary datasets.

- The external connector framework (described in Section 4.3) was designed to be general enough to let Pregelix work with any other Hyracks-based systems. Given that, two broadening directions would be interesting. First, we could build Pregelix connector implementations for a broader range of ecosystems, such as HBase [12], Hive [92], Cassandra [2], MongoDB [17], and Couchbase [3]. Those connectors would expand the landscape that Pregelix can work in. Second, we could reuse the current AsterixDB connector for other Hyracks-based systems, e.g., IMRU [85] or the Hadoop-compatibility layer [42]. We could then build a machine learning job entrance capability as well as a Hadoop job entrance capability in AsterixDB, which would further enrich the supported range of analytic tasks in AsterixDB.
- Temporary datasets (described in Section 4.2) are conceptually similar to materialized views [66] and to Spark RDDs (resilient distributed datasets) [108]. On one side, the internal mechanism of temporary datasets could be further extended to support materialized views in AsterixDB. Thanks to the indexing capability of AsterixDB, the materialized views would also be indexable. On the other side, similar to Spark RDDs, we could potentially build fault-tolerance mechanisms for temporary datasets for the case of slave machine failures, e.g., for recovering a lost partition from using its lineage data.

Bibliography

[1] BTC. http://km.aifb.kit.edu/projects/btc-2009/.

[2] Cassandra. http://cassandra.apache.org/.

[3] Couchbase. https://www.couchbase.com/.

[4] DataLake. https://en.wikipedia.org/wiki/Data_lake.

[5] Elastic Search. https://www.elastic.co/.

[6] Event Causality. https://en.wikipedia.org/wiki/Causality.

[7] Event Causality. https://plus.google.com/hangouts.

[8] Genomix. https://github.com/uci-cbcl/genomix.

[9] Giraph. http://giraph.apache.org/.

[10] Hadoop/HDFS. http://hadoop.apache.org/.

[11] Hama. http://hama.apache.org/.

[12] HBase. http://hbase.apache.org/.

[13] Hive+Giraph. https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920.

[14] Hivesterix. http://hyracks.org/projects/Hivesterix/.

[15] JSON. https://www.json.org/.

[16] Knowledge Graph. https://en.wikipedia.org/wiki/Knowledge_Graph.

[17] MongoDB. https://www.mongodb.org/.

[18] Pivotal. http://www.gopivotal.com/products/pivotal-greenplum-database.

[19] Pregelix. http://pregelix.ics.uci.edu.

[20] SQL. https://en.wikipedia.org/wiki/SQL.

[21] Teradata. http://www.teradata.com.

[22] The size of Facebook. https://en.wikipedia.org/wiki/Facebook/.

[23] The size of WWW. http://www.worldwidewebsize.com/.

[24] Twitter Usage Statistics. http://www.internetlivestats.com/twitter-statistics/.

[25] Two-phase Locking. https://en.wikipedia.org/wiki/Two-phase_locking.

[26] Vertica. http://www.vertica.com.

[27] VXQuery. http://vxquery.apache.org/.

[28] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. pages 174–185, 1995.

[29] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[30] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *PVLDB*, 7(10):841–852, 2014.

[31] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. pages 739–753, 2010.

[32] Spark-R. https://amplab-extras.github.io/SparkR-pkg/.

[33] The Distributed DataFrame Project. http://ddf.io/.

[34] The Mahout Project. http://mahout.apache.org/.

[35] The SparkSQL Project. https://spark.apache.org/sql/.

[36] The Tez Project. http://tez.apache.org/.

[37] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, pages 16–52, 1986.

[38] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[39] W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time java. In *International Conference on Embedded Software (EMSOFT)*, pages 289–305, 2001.

[40] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. pages 22–32, 2008.

[41] V. Borkar, Y. Bu, P. Carman, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: A data model-agnostic compiler backend for big data languages. In *SOCC*, 2015.

[42] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[43] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time java. pages 324–337, 2003.

[44] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[45] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[46] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. pages 363–375, 2010.

[47] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[48] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, page 3, 2012.

[49] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD Conference*, pages 457–468, 2012.

[50] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[51] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. pages 137–150, 2004.

[52] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[53] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[54] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[55] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. pages 59–70, 2008.

[56] S. Even. *Graph Algorithms.* Cambridge University Press, New York, NY, USA, 2nd edition, 2011.

[57] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[58] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 407–416, 2000.

[59] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *VLDB*, pages 209–219, 1986.

[60] D. Gay and A. Aiken. Memory management with explicit regions. pages 313–323, 1998.

[61] D. Gay and A. Aiken. Language support for regions. pages 70–80, 2001.

[62] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994.

[63] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[64] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[65] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. pages 282–293, 2002.

[66] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[67] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. pages 141–152, 2002.

[68] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. pages 73–84, 2004.

[69] I. Hoque and I. Gupta. LFGraph: Simple and fast distributed graph analytics. In *TRIOS*, 2013.

[70] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[71] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[72] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Architecture and Synthesis for Embedded Systems (CASES)*, pages 288–297, 2002.

[73] 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*, 5(8):716–727, 2012.

[74] H. Makholm. A region-based memory manager for prolog. pages 25–34, 2000.

[75] 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 Conference*, pages 135–146, 2010.

[76] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.

[77] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. pages 77–97, 2009.

[78] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[79] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. pages 245–260, 2007.

[80] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. pages 429–451, 2006.

[81] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. pages 1099–1110, 2008.

[82] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[83] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[84] R. Ramakrishnan and J. Gehrke. *Database Management Systems (3. ed.).* McGraw-Hill, 2003.

[85] J. Rosen, N. Polyzotis, V. R. Borkar, Y. Bu, M. J. Carey, M. Weimer, T. Condie, and R. Ramakrishnan. Iterative mapreduce for large scale machine learning. *CoRR*, abs/1303.3517, 2013.

[86] S. Salihoglu and J. Widom. GPS: a graph processing system. In *SSDBM*, page 22, 2013.

[87] SFrame. https://dato.com/products/create/docs/generated/graphlab.SFrame.html.

[88] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, 2001.

[89] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. pages 127–142, 2008.

[90] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD Conference*, pages 505–516, 2013.

[91] D. E. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *PVLDB*, 7(13):1405–1416, 2014.

[92] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[93] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[94] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lamda-calculus using a stack of regions. pages 188–201, 1994.

[95] Storm: dstributed and fault-tolerant realtime computation. https://github.com/nathanmarz/storm.

[96] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *SoCC*, page 5, 2013.

[97] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: a resilient distributed graph system on spark. In *GRADES*, page 2, 2013.

[98] G. Xu. Finding reusable data structures. pages 1017–1034, 2012.

[99] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. pages 174–186, 2010.

[100] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. pages 419–430, 2009.

[101] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.

[102] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. pages 151–160, 2008.

[103] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. pages 160–173, 2010.

[104] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. pages 738–763, 2012.

[105] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[106] D. Yan, J. Cheng, K. Xing, W. Ng, and Y. Bu. Practical pregel algorithms for massive graphs. In *Technique Report, CUHK*, 2013.

[107] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD Conference*, pages 517–528, 2012.

[108] 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 computingt. In *NSDI*, 2012.

[109] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research In Genome Research*, 18(5):821–829, 2008.

[110] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: a distributed framework for prioritized iterative computations. In *SoCC*, page 13, 2011.