Runtime Optimizations for Large-Scale Data Analytics

Jingjing Wang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Magdalena Balazinska, Chair

Bingni Burton

Alvin Cheung

Arvind Krishnamurthy

Dan Suciu

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Runtime Optimizations for Large-Scale Data Analytics

Jingjing Wang

Chair of the Supervisory Committee:
Magdalena Balazinska
Paul G. Allen School of Computer Science & Engineering

Large-scale data analytics is key to modern science, technology, and business development. Big data systems have emerged rapidly in recent years, but modern data analytics remains challenging due to application requirements. First, users need to analyze vasts amount of data generated from various sources; Second, analysis efficiency is critical in many scenarios; Third, most big data systems are built on top of new operating environments, such as clusters and cloud services, where resource management is important; Fourth, high-level, feature-rich programming interfaces are required to support a high variety of data and workload types.

In this dissertation, we present methods to improve system efficiency for large-scale data analytics. We investigate the problem in the context of three research projects, which address the same key problem: *optimization of analytical query execution*, in different ways. Specifically, these projects focus on *runtime* optimzation, which considers not only *static* information that is available prior to the actual execution, but more importantly, runtime information. We demonstrate, from these projects, that runtime optimzation can significantly improve overall system performance: it can lower query execution times, improve resource utilization, and reduce application failures.

We first present a full-stack solution for recursive relational query evaluation in shared-nothing engines. Users express their analysis using a high-level declarative language (a subset of Datalog with aggregate functions). Queries are then compiled into distributed query plans with termination guarantee. Multiple execution models for iterative queries are supported, including synchronous, asynchronous, and different processing priorities. Our evaluation shows that application properties determine which model yields the fastest query execution time.

Next, we present ElasticMem, an approach for automatic and elastic memory management for cloud data analytics applications. In clouds or clusters, a resource manager schedules applications in containers with hard memory limits, which requires accurate application memory usage estimation before launching containers. However, memory estimation for large-scale analytical applications is difficult, and inappropriate estimate can lead to failures and performance degredation. ElasticMem avoids pre-execution memory usage estimation by elastically allocating memory across containers during runtime. Experiments show that ElasticMem outperforms static memory allocation, leading to fewer query failures, lower garbage collection overheads, and lower query times.

Lastly, we present Deluceva, a system that dynamically optimizes neural network inference for video analytics. Many video analysis approaches apply neural network models trained on images directly to each video frame. While being easy to develop, these approaches do not leverage the rich temporal redundancy in videos, which can be used to further reduce model inference time. Deluceva accelerates model inference by dynamically selecting sufficiently significant temporal deltas to process for each video frame. Evaluation on three models and six videos shows that it can achieve significant performance gains with low errors.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to express my sincerest gratitude to my advisor Magdalena Balazinska. The way that she interacts with students and collaborators, her attitude and standards of academic research, and more generally, her wisdom on career and life, have set up great examples for me to observe, think, learn, and execute during these years. I am especially grateful to be consistently impacted by her perfectly balanced advising style: strict but also tolerant, optimistic but a believer of Murphy's law, patient but pushes when needed, leaving students space to grow but also providing detailed, reliable, hands-on mentoring. I hope that I have picked up these characteristics to some extent and hopefully will have the chance to apply them one day.

In addition to Magda, I would like to also thank my other "unofficial" advisors. I want to thank Dan Suciu for his thorough and deep understanding and extremely solid support on any problem relevant to theory. I would also like to thank Bill Howe and Alvin Cheung for bringing their perspectives and expertise from other sub-areas, and Arvind Krishnamurthy and Bingni Burton for being my committee members. During my internships, I also had the great pleasure and honor to work with researchers outside of UW. I would like to thank Surajit Chaudhuri, Sudipto Das, Bolin Ding, Vivek Narasayya, Manoj Syamala, and many others from Microsoft Research, for their support and advice on my project and also my career. Although not directly relevant to my PhD, I also want to thank Haixun Wang for bringing me into my first (formal) research project and for his mentoring.

I have been extremely fortunate to have been a part of the UW database group. During my years here, I received not only thorough training on my academic skills, but more importantly, an environment that encourages our initiatives, broadens our views, and nourishes genuine feedback, discussions, and friendship. I am also very grateful to my advisor, UW CSE, and all the funding agencies, for providing me with a stable and welcome environment to do research.

This chapter of my life would not have been completed without support from my friends. I would like to start with people from the database group: Maaz Bin Safeer Ahmad, Victor Teixeira de Ameida, Seung-Hee Bae, Tobin Baker, Walter Cai, Lee Lee Choo, Shumo Chu, Eric Gribkoff, Helga Guomundsdottir, Daniel Halperin, Brandon Hayes, Dylan Hutchison, Jeremy Hyrkas, Srinivasan Iyer, Shrainik Jain, Tomer Kaftan, Nodira Khoussainova, Paris Koutris, YongChul Kwon, Ryan Maas, Parmita Mehta, Dominik Moritz, Kristi Morton, Brandon Myers, Laural Orr, Jennifer Ortiz, Guna Prasaad, Dan Radion, Sudeepa Roy, Vaspol Ruamviboonsuk, Babak Salimi, Emad Soroush, Prasang Upadhyaya, Andrew Whittaker, Shengliang Xu, Cong Yan, and many others. It was my great pleasure and honor to complete this chapter of my life together with them at different times from different aspects: writing papers, doing code reviews, discussing ideas, asking for/giving feedback, sharing successes and failures, and having fun together. Besides database people, I got to know many people across CSE, UW, Seattle, and the research community during these years. It would be a long list if I listed all, so here I am only able to thank them all from my heart. I also want to thank friends that I met before or during college and are/were also pursuing PhD degrees at the same time: Chen Chen, Yizheng Chen, Wenhan Dai, Bailu Ding, Chengjun Fang, Haijie Gu, Qin Jia, Yanyan Jiang, Yuan Li, Yin Lou, Shuxiang Ruan, Chen Wang, Yue Wang, Wenlei Xie, Mohan Yang, Jiacheng Yang, Shuang Yang, Jiaqi Zhai, Jun Zhang, Tao Zou, and many others, for their companions on this journey. Finally, inspired by a technique from Halperin [101] and Khoussainova [128], I intentionally left out a few people who are very important to me. You know who you are.[1]

Finally, I would like to thank my parents. I began my PhD journey with motivations either explicitly or implicitly impacted by them and what they have been consistently doing, valuing, and believing in. Their unique ways of showing unconditional support: one stimulates me when I slack off, the other comforts me for my defeats, have always been my rock for this journey and beyond.

[1] I swear that it is not just for leaving myself some space.

Chapter 1
INTRODUCTION

The ability to manage, process, and analyze large-scale datasets is key to modern science, business, and technology. For example, astronomers today work with telescope images from sky surveys, such as LSST [148], that require the analysis of tens of terabytes of data per night; retailers, such as Walmart, use or build cloud solutions to cope with petabytes of transactional data every hour [36]; and city-wise traffic and security monitoring are now realities in countries such as the UK and China through the wide deployment of surveillance cameras [12, 22]. As the amount and variety of data continue to grow, large-scale data analytics continues to demonstrate its significant impact on people engaged in a wide range of professions.

In recent years, many new systems have emerged to help users analyze such large and varied datasets. These systems, often called *big data systems*, include commercial (e.g., Amazon Redshift [3], Vertica [34]) and open-source (e.g., Apache Spark [225], Apache Impala [129], TensorFlow [39]) engines. Big data analytics, however, remains challenging. First, many large-scale analytical applications are performance-critical and must thus meet strict requirements concerning system speed, efficiency, and robustness. These requirements are difficult to meet at large scale. Second, more computing resources are needed for large-scale processing, therefore systems designed to support these applications often run in new operating environments, such as clusters and cloud services, where resource management poses new, not fully understood problems. Third, data analyses involve a variety of data types (e.g., structured data, graphs, videos) and complex workloads: modern analytics includes not only traditional workloads, such as relational queries, but also graph analysis, machine learning and, more recently, deep learning.

Figure 1.1 illustrates a typical cloud-based data anlytics deployment: Users perform various types of analysis on multiple data types using big data systems deployed in the cloud. Executions

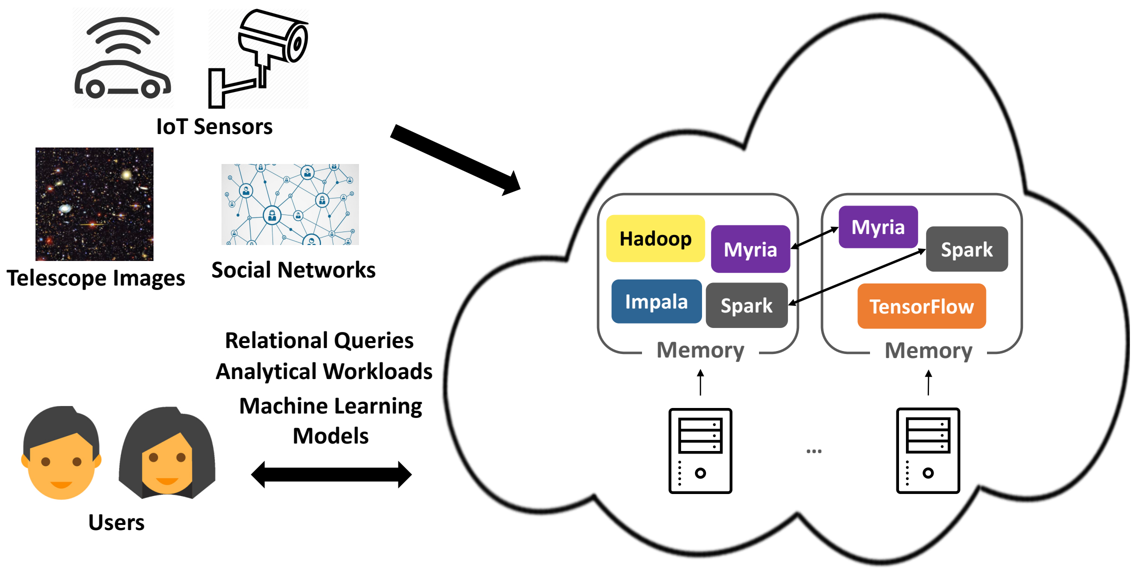


Figure 1.1: Overview of large-scale data analytics in the cloud.

are automatically optimized and distributed, cluster resources are shared between systems and applications, and various systems targeting at different workload types and with different performance characteristics may run together.

In this dissertation, we consider modern data management and analytics systems deployed in cloud computing environments as illustrated in Figure 1.1. We identify several limitations that cause these systems to underperform when executing modern data analytics workloads and propose innovative solutions to addressing those limitations. In this chapter, we first present the requirements of modern big data analytics applications that motivate our work and then given an overview of the key contributions of this dissertation.

1.1 Application Requirements

We focus specifically on the following important characteristics of modern data management and analytics applications:

- *Vast amounts of data.* Whether in the sciences or industry, users today must analyze large datasets. In addition to the use-cases above, other notable examples include the processing of large-scale outputs from pervasively-deployed devices: experts estimate that the Internet of Things (IoT) will consist of some 30 billion objects by 2020 [16]. Traditional relational databases supporting business analytics have also crossed into the petabyte range [36, 154]; and unstructured human-generated data, such as videos, photos, and tweets, is rapidly growing in volume. Youtube, for example, has over a billion users watching a billion hours of video each day [37].

- *Efficient analysis.* Efficiency is an important requirement in many modern data analysis scenarios. Many applications require efficient processing. For example, real-time decision-making applications, such as ride-sharing scheduling or autonomous driving, have strong requirements for low latency [13]. Users also require efficient processing. Data scientists need the ability to efficiently explore their data, and build summaries and models. The challenge in both cases lies in designing programming interfaces, systems, and infrastructures that perform fast analysis on large-scale datasets.

- *New operating environment.* Large-scale analytics requires large amount of resources, and new operating environments, such as clusters and cloud services [4], provide access to these resources. Many big data systems, such as Hadoop MapReduce [71], Spark [225], Flink [5], and Myria [213], are designed to run in clusters or in the cloud, where data and computation are automatically distributed for scalability and performance. In such environments, resources may need to be shared by multiple applications, systems, and users at the same time, but each execution unit may also need to be protected and isolated from the others. As the number of big data systems and use cases continue to grow, efficient resource management plays an important role in their cluster or cloud deployments.

- *Complexity of analysis.* User needs have dramatically changed over the past twenty years, and modern users increasingly require support to perform analysis over diverse data types. These types include not only traditional business data, but also images, videos, high-throughput sensors or devices, click streams, and social networks data. Further, the types of analyses that users seek to perform has also grown in sophistication, from traditional relational algebra operations to machine learning and linear algebra workloads. To support such diversity, high-level query languages with features, such as iterative constructs, user-defined functions, and domain-specific transformations, are necessary in many cases [213].

Motivated by these considerations, this thesis focuses on methods to improve the efficiency of large-scale data analytics. Specifically, we investigate the problem in the context of the following three research projects, which reflect big data analytics properties in different ways.

1.2 *Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines*

One distinguishing feature of modern analytics is that iterations occur in many workloads. Graph analytics is a prime example: shortest path, reachability, and connected components are all iterative algorithms. The need for iterative computations extends beyond graphs, though. With astronomy simulations, for example, a common type of analysis traces the evolution of galaxies in the universe by iteratively following simulated particles over time [145]. Many machine learning algorithms are also iterative. Users such as data scientists need to easily express iterative applications in high-level languages, and they also require efficient, scalable execution.

In response to this need, many existing data processing engines have either been extended to support iterative computations [55, 75, 227], built to execute both non-iterative and iterative queries [5, 225], or designed specifically for iterative computations [77, 157, 159, 179]. In these systems, however, *general-purpose*, *easy-to-express*, and *efficient*, distributed execution for iterative queries remains a key problem. Most parallel data processing engines support only *synchronous* iterations [42, 55, 152, 157, 225, 227], where all machines must complete one iteration before the next one begins. Such global synchronization barriers between iterations cause faster machines to wait for the stragglers, slowing down query evaluation. While some engines support parallel and asynchronous iterative processing, they are specialized for graphs [146,179,180,212]. Many general-purpose iterative query processing systems do not support declarative queries but instead require users to specify query plans directly [77, 159, 212]. Finally, some systems generate code [179, 180] and thus do not provide a complete and general-purpose data management system. Existing Datalog engines exist but are either single-node [21] or use MapReduce as a backend [84], which supports only synchronous iterations.

Based on our experience working with big data users in domain sciences through the University of Washington's eScience Institute [204], we find that none of the above approaches is sufficient. Modern engines should provide the following five capabilities for iterative computations. First, query evaluation should be performed in parallel in shared-nothing clusters to ensure scalability. Second, iterative query evaluation should be incremental, so that each iteration computes a change only to the final solution rather than recomputing the entire result each time. Third, the system should provide a variety of iteration models (synchronous, asynchronous, prioritized) because, as we will show, different problems necessitate different evaluation strategies. Fourth, engines must support a broad variety of application domains, not only graphs. Finally, users should be able to specify their computation using a declarative query language for ease of use. These features should be implemented in an efficient and fault-tolerant manner.

In Chapter 2, we develop such a new query evaluation approach [216] that supports *incremental*, *asynchronous* or *synchronous* evaluation of iterative queries in *shared-nothing* clusters. Users specify their queries declaratively *in a subset of Datalog with aggregation*, and queries are compiled into distributed execution plans with multiple possible execution models. Taking advantage of asynchronous execution, our approach also features a lightweight failure-handling solution. Finally, it requires only *small extensions* to a shared-nothing query processing system, making it generally well suited for implementation in big data systems.

1.3 ElasticMem: Elastic Memory Management for Cloud Data Analytics

Big data systems, such as Spark [225], Myria [213], and many others [3, 5, 6, 129, 159], are typically designed to operate in clusters or cloud environments, where shared pools of system resources and services can be acessed with minimal management effort, to leverage abundant resources for large-scale analytics. In such environments, computing resources are typically shared by many queries (also called "applications"), and even many systems executing in the same cluster simutaneously. A resource manager [109, 207] is commonly used in such shared clusters. Modern resource managers rely on containers (*e.g.*, YARN [207], Docker [9], or Kubernetes [19]), which isolate applications that share the same machine and provide hard resource limits. Application resources are both constrained and protected by the containers.

In Chapter 3, we focus in particular on *memory* allocation because in-memory processing is key to efficient large-scale analytics, yet memory remains an expensive resource. Many modern data analytics systems strive to maximally utilize memory. However, container-based scheduling has limitations in this regard: when an application needs to run, it must estimate its resource requirements and communicate them to the resource manager. The latter then decides whether or not to schedule the application based on the amount of available resources. The challenge, however, is the difficulty of estimating the memory need of a data analytics application *before* executing it, since its needs may depend on multiple runtime factors, including the cardinalities of intermediate results, known to be hard to estimate [118, 137]. Inaccurate memory usage estimates can cause either poor performance or poor resource utilization.

To address these problems, we develop a new approach, called *ElasticMem* [214, 215], where data analytics applications execute in separate containers, but the resource manager *elastically* adjusts the memory allocated to these containers. ElasticMem focuses on Java-based containers and large-scale analytical applications. First, it relaxes the rigid design of JVM hard memory limits by modifying it to enable dynamic memory layout adjustment during runtime. Second, ElasticMem includes performance models for analytical queries to measure the impact of possible memory allocation decisions on executing queries. Third, it has an algorithm to allocate memory across multiple applications during runtime towards a global objective function. As a result, ElasticMem substantially reduces application failure, garbage collection time, and query time for all applications subject to the physical limit of the total amount of memory in the cluster.

1.4 Deluceva: Delta-Based Neural Network Inference for Fast Video Analytics

Modern users increasingly need support analyzing more diverse types of data, particularly videos. Additionally, the type of processing that users perform has become more sophisticated and commonly includes machine learning algorithms and, most recently, deep learning. As a result, multiple new video data management and analytics systems have emerged [113, 150, 226]. Unlike traditional video database management systems [45], these systems let users perform complex video analytics tasks at large scale.

In Chapter 4, we focus specifically on video analytics applications that use deep learning at their core. Deep learning is a fundamental component of today's video analytics applications [113, 226]. Since videos can be treated as streams of images, most state-of-the-art video analysis approaches directly apply neural network models developed for single images to video frames and use these per-frame results for further processing [105, 124, 125, 171]. While it is possible to perform vision tasks on video frames directly, the cost of running image neural network models is high and continues to grow as models become more complex and inputs become larger. Representative object detection models run at speeds of less than one frame [172] to 90 frames [171] per second on high-end GPUs with low-resolution inputs (*e.g.*, 300×300). Higher resolution images and larger models are needed for more accurate and complex analysis [114], such as to detect small obstacles in autonomous driving [67] because many accidents are caused by road debris or hazardous cargo [26, 31].

To address this limitation, we develop Deluceva, a system that optimizes live video analysis. Deluceva accelerates a given neural network model's inference on video frames by processing *significant deltas* only, instead of processing full images.[1] It consists of three components. First, it includes efficient tensor operators that process delta inputs encoded with a sparse representation instead of processing full frames. Second, to save computation by delta-based processing, it processes only deltas that are sufficiently significant for each model evaluation. The criterion of being significant is dynamically adjusted based on error feedback. Third, whether each operator's sparse or dense variant is more efficient depends on runtime factors, so an optimal network may consist of operators of both types. Deluceva includes performance models for both types. It builds those models by profiling operators offline and constructing model variants accordingly for each model evaluation. We implement our approach in TensorFlow [39] and evaluate it on representative models and test videos. Our approach outperforms the original model inference by up to 79% in terms of runtime while keeping object detection errors low.

[1] Our focus is on accelerating model inference instead of model training.

1.5 Summary of Contributions and Thesis Outline

Although the three projects at the heart of this dissertation focus on different aspects of system efficiency for large-scale data analytics, their core techniques address the same key problem: *optimization of analytical query execution*. Traditional techniques, such as database query optimization, perform *static* optimization, where the optimal execution plan is determined statically prior to the actual execution. However, our experience with the preceding projects led us to observe that static optimization is not always sufficient or optimal when runtime information must be considered. When we lack access to such information, statically optimized execution may lead to suboptimal results, such as redundant computation, unexpected query failures, and resource overprovisioning. The complexity of big data further increases the unpredictability of runtime execution. Motivated by these considerations, this thesis addresses the problem of *runtime* optimzation for large-scale data analytics from multiple perspectives.

In summary, this thesis addresses runtime optimizations for large-scale data analytics, making the following contributions:

- **Full-stack solution for multiple execution models of iterative relational queries.** We develop a full-stack approach [216] for large-scale iterative data analytics that combines the benefits of many existing systems: users express their analysis in recursive Datalog with aggregation, which simplifies the expression of data analysis tasks from a variety of application domains. The system executes the analysis using parallel query plans that require only small extensions to an existing shared-nothing engine yet deliver the full power of incremental synchronous and asynchronous query evaluation even for query plans with multiple recursively computed relations.
  We implement our approach in the Myria big data management system and service [23, 102, 213]. We empirically evaluate how small changes in the query evaluation strategy can lead to large performance differences for different types of queries and characterize when each approach leads to the lowest query runtime. We also empirically study the interplay between iterative processing method, iterative query, and failure handling method. We find that no single iterative execution strategy outperforms all others; rather, application properties must drive method selection

(Chapter 2).

- **Dynamic, elastic memory management for cloud data analytics.** We develop ElasticMem [214, 215], a technique for the automatic and elastic memory management of big data analytics applications running in shared-nothing clusters. ElasticMem includes a technique to dynamically change JVM memory limits, an approach to model memory usage and garbage collection cost during query execution, and a memory manager that performs actions on JVMs to reduce total failures and runtimes.

  We evaluate ElastcMem using TPC-H queries [29] on Myria [23, 102, 213] against containers with fixed memory limits. Our approach outperforms static allocation in terms of query failure, garbage collection time, query execution time, and overall resource utilization (Chapter 3).

- **Delta-based neural network inference for fast video analytics.** We develop Deluceva, a system that optimizes live-video analysis using deep learning by applying incremental and approximate computation techniques. Our approach includes: (1) a novel method for incremental deep network inference with new, specialized operators, (2) two algorithms to dynamically adjust the amount of data processed to minimize runtime subject to achieving a target quality, and (3) a new method to generate mixed networks with sparse and dense operators.

  We implement our approach in TensorFlow [39] and evaluate it on three representative object detection models and six videos. The evaluation shows that our approach outperforms the original model inference by up to 79% in terms of total compute resources with F_1 errors below 0.1 (Chapter 4).

We discuss related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

ASYNCHRONOUS AND FAULT-TOLERANT RECURSIVE DATALOG EVALUATION IN SHARED-NOTHING ENGINES

In this chapter, we focus on the problem of declarative specification and efficient execution of iterative queries. The work presented in this chapter appeared at VLDB'15 [216]. As we described in Section 1.2, iterative computations commonly occur in modern data analysis applications, such as graph analytics, machine learing, and data science applications. While traditional methods to express and evaluate iterative queries exist [169, 175], they only provide limited expressiveness and usually assume single-machine execution, which is insufficient to handle the complexity and scale of modern analytics. We observe that newer systems also have important limitations. In particular, none [5, 42, 55, 75, 77, 157, 159, 179, 212, 225, 227] offer the combination of declarative, easy-to-use query languages, distributed execution, fault-tolerance, and support for both synchronous and asynchronous execution.

We address the above needs by developing a full-stack solution [216] to efficiently express and evaluate iterative relational queries in shared-nothing architectures. With our approach, users can easily express their queries using a declarative query language: a subset of *Datalog with aggregate functions*. The queries are then compiled and optimized into distributed in-memory execution plans. Our approach support multiple execution models for iterative queries: synchronous, asynchronous, and prioritized processing. For asynchronous computation, our approach also features a lightweight failure-handling solution with multiple optimization options.

We implement our approach in Myria [213] and evaluate it using three iterative applications: GalaxyEvolution (Figure 2.1, Figure 2.2), which computes the evolution history of galaxies in an astrophysical simulation; Connected Components (Figure 2.5), which outputs the connected components of an undirected graph; and Least Common Ancestor (LCA, Figure 2.6), which derives the least common ancestor of pairs of publications from a citation graph. The key questions that we answer are: (1) what class of positive Datalog queries with aggregation can be evaluated recursively and asynchronously in shared-nothing systems (*i.e.*, the aggregate function can be evaluated inside the iteration), (2) which execution model to choose to achieve the best performance and how to generate a distributed query plan with termination guarantee, and (3) which fault-tolerance method imposes a low overhead yet achieves fast failure recovery for each execution model. Specifically, we make the following contributions:

- We develop a query-plan based approach for efficient, incremental processing of iterative computations in a parallel, relational engine. Our approach supports iterative computations with aggregations and with multiple output results. It supports both synchronous and asynchronous query evaluation, and different processing priorities. Our approach is easily implementable as it only requires a small set of extensions to an existing shared-nothing engine (Section 2.2.2 and Section 2.2.3).
- We enable users to specify declarative queries in a subset of Datalog with aggregation extended from prior work [180] and present an algorithm to automatically convert these programs into our new query plans (Section 2.2.1).
- We empirically evaluate different query evaluation methods on iterative applications from the astronomy, social networks, and bibliometrics domains and on real data from these domains. We characterize when and why each query evaluation method yields the lowest query runtime (Section 2.5.2).
- We study the interplay of iterative query execution methods, iterative queries, and fault-tolerance techniques. We evaluate which failure-handling method yields the fastest recovery in the presence of failures while imposing a small overhead without failures (Section 2.3 and Section 2.5.3).

Exploring a broad set of applications and techniques, we find that no single execution strategy wins in all cases as method selection must serve application properties. The wrong choice of execution technique can increase CPU utilization and network bandwidth by a factor of 6×. Memory usage can increase by a factor of 2× (Section 2.5). The key lies in the amount of intermediate data being generated during query evaluation.

2.1 Background

In this section, we briefly review how to express iterative queries with aggregation in Datalog and present existing query evaluation methods for iterative queries in shared-nothing systems.

2.1.1 Iterative Queries in Datalog

Iterative queries are naturally expressed in Datalog as this language supports recursion. As an example, Figure 2.1 shows an iterative Datalog program called GalaxyEvolution. The program computes the history of a set of galaxies in an astrophysical simulation. The history of a galaxy is the set of past galaxies that merged over time to form the galaxy of interest at present day. The input table, `Particles(pid,gid,time)`, holds the simulation output as a set of particles, where `pid` is a unique particle identifier, and `gid` is the identifier of the galaxy that the particle belongs to at time, `time`. The `gid` values are unique only within their timesteps, `time`, but a particle retains the same `pid` throughout the simulation.

The program takes the form of a set of rules. Each rule has a head, an implication symbol `:-`, and a body. The first rule computes relation `Edges`, which contains the number of shared particles for every pair of galaxies at adjacent timesteps. The second and the third rules compute `Galaxies`, which is the set of earlier galaxies that have merged to form the galaxies of interest. The second rule states that a galaxy from `GalaxiesOfInterest` at present day (*i.e.*, `time=1`) is a part of the ancestry. The third rule states that a galaxy `(time+1,gid2)` is also part of the ancestry if an adjacent galaxy `(time,gid1)` is part of the ancestry, and the number of shared particles between the two galaxies is above the threshold, `thresh`. This last rule is recursive because the relation Galaxies appears both in the head and in the body of the rule.

In Datalog programs, base data is referred to as Extensional Database predicates (EDBs) (*e.g.*, `Particles`). Derived relations are Intensional Database predicates (IDBs) (*e.g.*, `Edges` and `Galaxies`). EDBs occur only in bodies, while IDBs appear in the heads of the rules and can also

```
Edges(time,gid1,gid2,$Count(*)) :- Particles(pid,gid1,time),
                                   Particles(pid,gid2,time+1)          (1)
Galaxies(1,gid)                 :- GalaxiesOfInterest(gid)             (2)
Galaxies(time+1,gid2)           :- Galaxies(time,gid1),
                                   Edges(time,gid1,gid2,c),c >= thresh (3)
```

Figure 2.1: Datalog program for the GalaxyEvolution application. The inputs are relation Particles(pid,gid,time), which contains the output of an astrophysical simulation, and a set of galaxies of interest GalaxiesOfInterest(gid) at time=1.

appear in the bodies.

Without aggregations and without negations (*i.e.*, in the case of conjunctive queries), the result of a Datalog program is its least fixpoint, which can be computed by repeatedly evaluating the rules in any order until no new facts (*i.e.*, tuples) are found.

When a positive Datalog program includes aggregations, as in our example, it is evaluated in a stratified manner, which means that the program is evaluated one subset of rules at the time. More specifically, the aggregate function is evaluated only after all predicates that form the right hand-side of the corresponding rule have been fully evaluated (*i.e.*, have reached their fixpoint). In our example, the condition holds trivially since the rule with the `$Count` aggregate has only an EDB in its body rather than one or more IDBs. Similarly, the rules that use the aggregate value are blocked until the aggregate has been computed. In the example, rules (2) and (3) are not evaluated until rule (1), which contains the aggregate function `$Count`, has been evaluated.

In this work, we focus on positive Datalog programs (no negated predicates) with aggregate functions that only occur in IDBs. Given an IDB with a set of grouping attributes v and aggregate functions f, the semantics are those of group by aggregation: For all tuples that satisfy the body of the rule, the rule evaluates the value of each function f for each unique combination of values of v.

2.1.2 Shared-Nothing Iterative Processing

There exist three common approaches to evaluate iterative query plans in shared-nothing systems:

- *Bulk synchronous*: Each iteration computes a completely new result from the result of the previous iteration. A synchronization barrier separates each iteration.
- *Incremental synchronous*: Each iteration computes a new result, then compares it with the result of the previous iteration and feeds only the delta to the next iteration. This approach corresponds to semi-naïve Datalog evaluation.
- *Incremental asynchronous*: New facts are continuously discovered without coordination between operator partitions and without any synchronization barriers.

Consider GalaxyEvolution from Figure 2.1. During the evaluation of rules (2) and (3), with bulk synchronous execution, each iteration i takes all the galaxies reachable from `GalaxiesOfInterest` within $i-1$ timesteps as input, then joins them with `Edges` to get all the galaxies reachable within i timesteps. With the incremental synchronous evaluation, the input of iteration i is only the set of galaxies that are reachable with no less than $i-1$ timesteps. The output contains only newly discovered reachable galaxies. With asynchronous execution, operator partitions continuously discover and communicate reachable galaxies without synchronization barriers.

2.2 Asynchronous Evaluation of Datalog with Aggregation

In this section, we present our asynchronous evaluation technique for recursive queries. We first introduce an extended class of aggregate functions that can recursively be evaluated in Datalog programs (Section 2.2.1). We then show how to generate query plans for the asynchronous and parallel evaluation of Datalog programs with this extended class of recursive aggregates (Section 2.2.2). Finally, we discuss optimizations that can significantly affect query performance and example applications that illustrate these performance trade-offs (Section 2.2.3).

2.2.1 Recursive Bag-Monotonic Aggregation

As described in Section 2.1.1, the non-recursive method to evaluate a Datalog program with aggregation consists in evaluating an aggregate function only after all its input IDBs have converged to their fixpoints. Furthermore, if an IDB with an aggregate function is used in the body of another rule, the evaluation of that rule also blocks until the aggregate function is evaluated.

In some cases, recursively evaluating these aggregates can speed-up convergence by pruning unnecessary partial results early. In contrast, the blocking strategy must evaluate each IDB in full, which may yield worse performance as we show in Section 2.5.2. The SociaLite work [180] has shown that a small class of aggregates, *meet* aggregates, can be evaluated recursively. These aggregate functions are associative, commutative, and idempotent binary operations defined on a domain S. These operations, denoted with $\wedge$, induce a partial order $\preceq$ on S, which is defined as: $\forall x, y \in S, x \preceq y$ if and only if $x \wedge y = x$. The result of the function on any two elements is the greatest lower bound with respect to this partial order. `$Min` and `$Max` are two examples of meet operations. Furthermore, if a Datalog program comprises a meet aggregate function defined on a finite set, and the rest of the program is monotonic with respect to the partial order induced by the aggregate, then the iterative evaluation of the rules converges to the greatest fixpoint. Finally, these programs can be evaluated incrementally [180] and asynchronously [179].

As in SociaLite, we support meet aggregates, which already enables us to express a subset of our target applications (see Section 2.2.3) . We observe, however, that many applications require aggregates other than meet aggregates, yet can still benefit from the recursive evaluation of those aggregates. Typical examples of these aggregates are `$Count` and `$Sum`. These aggregates are commonly used in analytical applications yet are not meet aggregates because they are not idempotent.

GalaxyEvolution is one example application that can benefit from the recursive evaluation of a `$Count` aggregate, which is illustrated by the Datalog program from Figure 2.2. This program computes the same result as the one in Figure 2.1 but uses different rules, which involve recursive aggregates. Here, the `Edges` IDB depends on the recursively defined `Galaxies` IDB. As a result, `Edges` and its `$Count` aggregate must be recursively evaluated as well. We show in Section 2.5.2 that the recursive version of the application can significantly reduce the run time because it avoids the computation of unnecessary tuples in `Edges`. The clustering algorithm DBSCAN [76] is another example application that can benefit from the recursive evaluation of a `$Count` aggregate (used during the density estimates).

The above examples and other similar examples that we encountered while working with scientists at the University of Washington motivate us to extend the notion of recursive aggregates to a broader class of aggregate functions: We show that it is possible to recursively evaluate aggregate functions that are commutative, associative, and *bag-monotonic* (but not necessarily idempotent). Examples of bag-monotonic aggregates include `$Count`, `$Sum`, which are not idempotent, and also include `$Min` and `$Max`. We start with a definition of a bag-monotonic aggregate:

Definition 2.2.1. *Let S be a set of bags of tuples, and $x, y \in S$. A partial order $\preceq$ on S is defined as: $x \preceq y$ if and only if $x \subseteq_{bag} y$* [1]*. An aggregate function $a : S \rightarrow V$ is* `bag-monotonic` *with respect to a partial order $\preceq$ defined on V, if for any $x, y \in S$ such that $x \preceq y$, we have $a(x) \preceq a(y)$.*

We can now define a Datalog program with a recursive, `bag-monotonic` aggregate. Let $g : T \rightarrow W$ be the function that takes a bag of tuples $R \in T$, does a group by on the set of grouping attributes, then applies the aggregate function a to each group k to generate a set of tuples $U = \{(k, v) \ldots\}, v \in V, U \in W$. The partial order on T is defined as: $R_1 \preceq R_2$ if and only if $R_1 \subseteq_{bag} R_2$. The partial order on W is its Hoare order: $U_1 \preceq U_2$ if and only if $\forall(k, v) \in U_1$, $\exists(k', v') \in U_2$, $(k, v) \preceq (k', v')$, where $(k, v) \preceq (k', v')$ if and only if $k = k', v \preceq v'$. We refer to the rest of the program as $f : W \rightarrow T$, and require f to be monotonic with respect to the order defined on T and distributive. The whole program is then defined as the recursive application of the function $(f \circ g)$. Starting with a set of empty bags of tuples R_0, for each $i \geq 0$, we have:

$$U_{i+1} = g(R_i), R_{i+1} = f(U_{i+1}).$$

We illustrate the definition using Figure 2.2. In this program, the `$Count` aggregate computes the number of particles shared between any pair of galaxies, `(gid1,gid2)`, at adjacent timesteps. As the rules are evaluated, more particles can be found to satisfy the body of rule (3). As a result, the bag of particles that serves as input to `$Count` grows. Each bag is a superset of the previous bag, thus only causes `$Count` to be computed on supersets of the previous inputs. Since `$Count` is

[1] $\subseteq_{bag}$ is bag containment, $x \subseteq_{bag} y$ if and only if for any tuple t that appears n times in x, t also appears m times in y, and $n \leq m$.

bag-monotonic, the output of the aggregate only increases. Furthermore, once a new pair of galaxies is discovered, no new tuples can cause the pair to be removed. Notice that if the condition in rule (2) was changed to `c < thresh`, the program would no longer be monotonic with respect to the partial order defined on the input to the aggregate operator.

Finally, we use a similar approach as in SociaLite to show that the naïve evaluation of such a program converges to a fixpoint, and that its semi-naïve evaluation is equivalent to the naïve evaluation. Since R_0 consists of sets of empty bags, we have $R_0 \preceq (f \circ g)(R_0)$. Using mathematical induction, if T is a finite set, then there must exist a finite n such that

$$R_0 \preceq (f \circ g)(R_0) \preceq \cdots \preceq (f \circ g)^n(R_0) = (f \circ g)^{n+1}(R_0),$$

where $(f \circ g)^n(R_0)$ is the greatest lower bound of the program.

The process of semi-naïve evaluation is the following. Starting from $R'_0 = R_0$, for each $i >= 0$, we have:

$$\begin{aligned} U'_{i+1} &= g(R'_i), \\ \Delta_{i+1} &= U'_{i+1} - U'_i, \\ R'_{i+1} &= R'_i \cup f(\Delta_{i+1}). \end{aligned}$$

We use mathematical induction to show that $U'_i = U_i$.

Basis: $U'_1 = g(R'_0) = g(R_0) = U_1$.

Inductive: assuming $U'_k = U_k$ for all $k \leq i$, then we have:

$$\begin{aligned} U'_{i+1} &= g(R'_i) = g(R'_{i-1} \cup f(\Delta_i)) \\ &= \dots \\ &= g(f(\Delta_1) \cup f(\Delta_2) \cup \cdots \cup f(\Delta_i)) \\ &= g(f(\Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_i)) \\ &= g(f(U'_i)) = g(f(U_i)) = U_{i+1}. \end{aligned}$$

```
Galaxies(1,gid)                  :- GalaxiesOfInterest(gid)                              (1)
Galaxies(time+1,gid2)            :- Galaxies(time,gid1),
                                    Edges(time,gid1,gid2,c),c >= thresh                  (2)
Edges(time,gid1,gid2,$Count(*)) :- Galaxies(time,gid1),
                                    Particles(pid,gid1,time),Particles(pid,gid2,time+1)  (3)
```

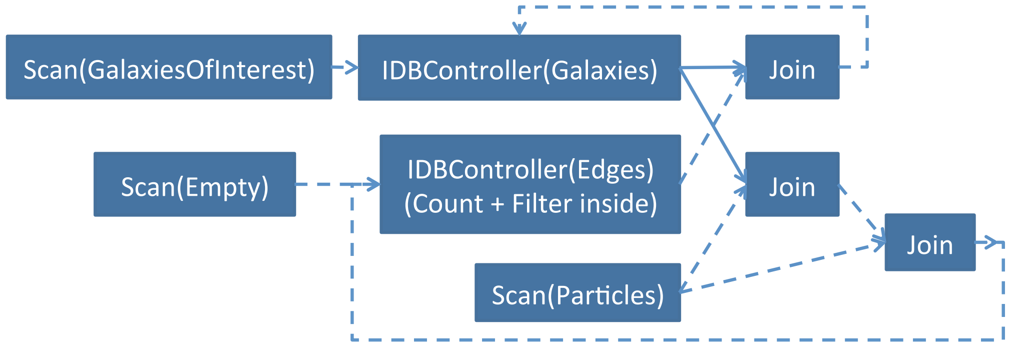


Figure 2.2: Datalog program for the GalaxyEvolution application using a recursive aggregate (top). Query plan for this application (bottom). Dashed lines indicate shuffling over the network. Note that: 1. we push the selection into the IDBController as an optimization, 2. since the Edges IDB does not have an initial input, we link a Scan, which reads an empty relation, to initialize the IDBController for Edges.

2.2.2 *Parallel and Asynchronous Evaluation*

In this section, we show how to translate a Datalog program with bag-monotonic recursive aggregates into a query plan that can be executed *asynchronously* and *incrementally*. Our approach can be implemented in a broad class of big data management systems with only small extensions that we present in this section. Our approach then enables the asynchronous evaluation of even complex query plans with multiple recursive IDBs. The systems that we target are shared-nothing, dataflow, analytical engines, in which operators and data are horizontally partitioned across worker processes, and data can be pipelined from one operator to the next without going to disk and without synchronization barriers. Examples of such systems include Flink [5, 77], Dryad [120], parallel database systems [10, 87], and also our own system, Myria [102]. Spark [225] could also benefit from our approach if it was extended with pipelined data shuffling, while MapReduce [71] serves as a counterexample. Note that these specific systems already have their own approach to iterative processing (see Section 5.1). We use them as examples of the class of systems to which our approach also applies.

Recursive Query Plans

The incremental evaluation of recursive Datalog programs with bag-monotonic aggregate operators requires query plans that perform several functions: (1) At each iteration, the query plan must compute new facts based on the incremental changes to the states of the recursive IDBs since the last iteration. In the example from Figure 2.2, each iteration discovers new `Galaxies` tuples and new qualifying pairs of `Particles` tuples that join together. (2) The query plan must then update the state of the recursive IDBs based on the new facts. This state update may require the computation of the aggregate functions if present. (3) The query plan must compute and output the changes to the IDB states since the last iteration, such as the newly computed `Edges` and `Galaxies` tuples. (4) The plan must detect whether all IDBs have reached a fixpoint. We propose to use regular relational operators for (1). We encapsulate functions (2) and (3) into a new operator that we call IDBController. We introduce a second operator, the TerminationController, to check (4).

As shown in Figure 2.2, an IDBController has two input children. One child is the initial state of the IDB, which is not recursive. For `Galaxies`, this input is the relation `GalaxiesOfInterest`, although it can also be empty, as for `Edges`. The other child is the recursive input for the new tuples that are generated during the iterations.

We let the IDBController perform the group-by and aggregation within itself to avoid generating and moving unnecessary tuples between operators. Instead, the IDBController computes directly the aggregated state. Additionally, while the IDBController accumulates the complete IDB state, it outputs only changes to that state in order to support incremental evaluation.

An IDBController has two execution modes: synchronous and asynchronous. In synchronous mode, the IDBController first fully consumes its initial input, initializes the state of its IDB, and outputs that state. Second, it accumulates tuples from the recursive input until the end of one iteration. It then updates the state of its IDB and outputs changes to that state for the next iteration. In asynchronous mode, the IDBController consumes input tuples on either input as they become available. For each input tuple, it updates the state of the corresponding group, and outputs the new

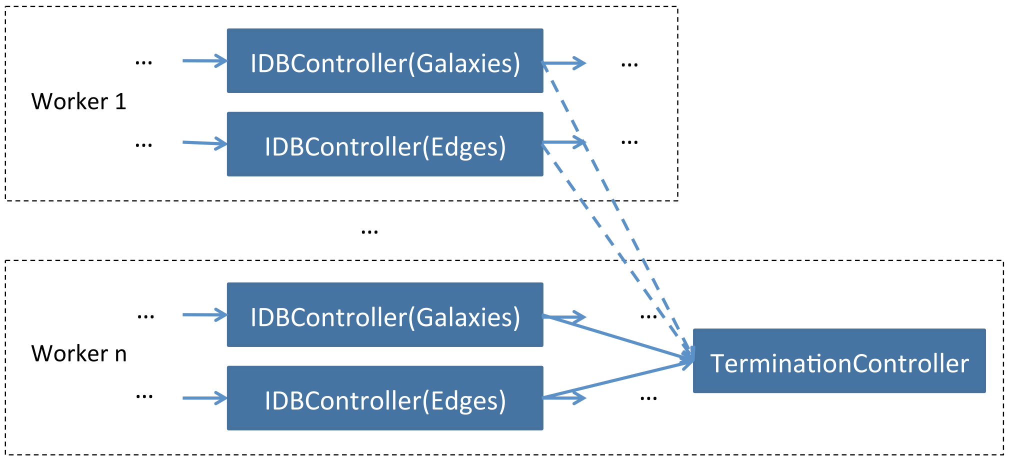


Figure 2.3: GalaxyEvolution query plan with IDBControllers and an TerminationController. Other operators are omitted.

aggregate value if it has changed. We explore the performance implications of each execution mode in Section 2.5.2. In the rest of this subsection, we focus on the asynchronous mode.

There is one TerminationController for each iterative computation. This operator collects periodic messages from all IDBControllers indicating whether the operators produced any new tuples since the last message. We further discuss the details of the TerminationController and fixpoint detection in Section 2.2.2.

Figure 2.3 illustrates the positions of the two operators in a query plan by showing the query plan produced for GalaxyEvolution. In all other figures, we omit the TerminationController (and the parallelism) when showing query plans.

Given these two special operators, we are now able to translate a positive Datalog program into an asynchronous recursive query plan, as shown in Algorithm 1. Briefly, the procedure translates the head of each rule into an IDBController, the body of each rule into a relational query plan. It then connects the appropriate inputs and outputs. As an example, Figure 2.2 shows the generated query plan for GalaxyEvolution, where the two IDBControllers recursively depend on each other.

Lightweight Termination Check

Evaluating a recursive query asynchronously raises an important issue: how to decide if the program has terminated. Since there are multiple IDBs being evaluated on multiple machines, and messages are sent through the network with possible delays, we need a protocol to guarantee a correct termination.

To detect termination, we propose a lightweight protocol that only requires small extensions to operators. First, we extend all operators with the ability to propagate a special message called *end-of-iteration* (EOI), based on the following three rules. Later in this section, we show how these messages serve as markers to determine that the fixpoint has been reached.

1. An IDBController generates the first EOI when its initial input has been fully consumed, and all the following EOIs when it receives an EOI from its iterative input.
2. An operator which is not an IDBController generates an EOI when it has received at least one EOI from each of its children since the last time it generated an EOI.
3. The internal design of an operator needs to ensure that, after it generates an EOI, it does not generate more tuples until it has fetched some new tuples from its children.

Additionally, an IDBController sends a message to the TerminationController every time it generates an EOI. Each message is a triple (R, α, b), where b is a boolean value indicating whether the IDBController for relation R on worker α generated any new tuples since the last message. The TerminationController maintains one relation for each IDB with one column per worker. Whenever it receives a message from a worker, the TerminationController appends the boolean value to column α in table R as shown in Figure 2.4.

When the TerminationController finds a *full-false row*, in which which all attribute values in all tables are `false`, the TerminationController signals that the query has completed. To ensure that the above condition correctly identifies the termination of the query, we need to prove two lemmas:

Lemma 2.2.1. *When the query terminates, there exists a row n such that any row i with $i \geq n$ is a full-false row.*

Proof. The above lemma follows directly from the rules in the protocol: operators output an EOI in

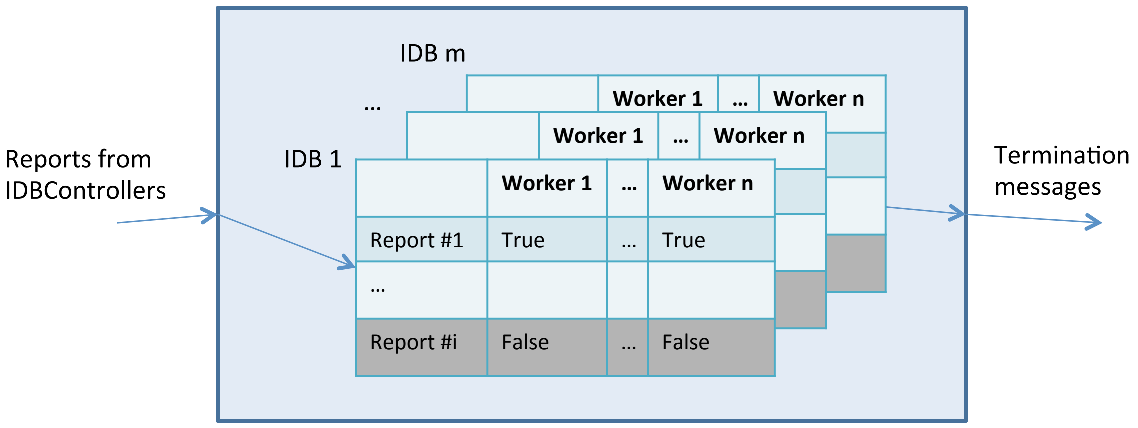


Figure 2.4: The internal state of an TerminationController

response to receiving EOIs on all their children and there is no termination condition for this process. Starting from some time t, if the query will not generate any more data, then all the following messages must be `false`. $\square$

Lemma 2.2.2. *If row k is a full-false row, then any row i, $i > k$, is also a full-false row.*

Proof. We prove the lemma by contradiction. Consider that operators produce EOIs with increasing sequence numbers (in our protocol, EOIs need not be numbered). We use $\text{EOI}_{t,A}$ to denote the t-th EOI produced by IDBController A. A outputs $\text{EOI}_{0,A}$ after consuming its entire initial input. Because an operator only propagates an EOI after receiving *at least one* EOI from each of its children, A will be able to produce $\text{EOI}_{1,A}$ only after all IDBControllers B that produce data consumed by A have produced their $\text{EOI}_{0,B}$. Furthermore, we use s_1 to denote the largest EOI sequence number that A has received as input before generating $\text{EOI}_{1,A}$, then $s_1 \geq 0$. [2] By induction, A outputs $\text{EOI}_{i+1,A}$ only after any recursively connected IDBController B has produced $\text{EOI}_{s_{i+1},B}$, and $s_{i+1} \geq i$.

Consider the case where k is a *full-false row* but there exist some `true` cells following row k. Consider t_a, the earliest tuple that was generated among all the `true` cells after row k. By rule 3 in the protocol, t_a was generated in response to some other tuple t_b and, by definition of t_a, t_b must

[2]To see why A can receive $\text{EOI}_{s_1,B}$ with $s_1 \geq 0$ before generating $\text{EOI}_{1,A}$, consider the case where B generates data consumed by A but is itself independent from A.

belong to a cell before row k. On the other hand, since $\mathrm{EOI}_{s_k,B}$ was generated before $\mathrm{EOI}_{k,A}$ based on the above induction and t_a goes after $\mathrm{EOI}_{k,A}$, then t_b must go after $\mathrm{EOI}_{s_k,B}$. Since $s_k \geq k - 1$ and k is a *full-false row*, we know t_b must live in a cell after row k. Thus the lemma is proven by contradiction. $\square$

2.2.3 Execution-Time Optimizations

Iterative query processing with aggregation is amenable to several execution time optimizations. Importantly, as we show in Section 2.5.2, selecting different execution strategies for the same query plan can significantly impact performance.

The first optimization is the decision to execute a query either *synchronously or asynchronously*. We use `sync` and `async` to denote these two execution modes respectively. We find that it is not the case that the latter always outperforms the former for applications that tolerate asynchrony.

Asynchronous processing has the benefit of resilience against uneven load distribution because workers process data without synchronization barriers. This benefit, however, can be offset by a larger amount of intermediate result tuples generated during execution. We demonstrate experimentally that, each combination of iterative application and execution strategy (`sync` or `async`), can generate a different number of intermediate result tuples, which significantly affects performance. The `sync` and `async` execution modes are supported by the IDBControllers as described in Section 2.2.2. The system sets the execution mode when initializing these operators for a query.

A second query execution strategy choice that can yield dramatically different numbers of intermediate result tuples is the *join pull order*. Binary operators, such as joins, can consume their input data in several ways. One approach is to fully consume one of the inputs before fetching any data from the other one. Alternatively, a join can pull from both children with or without preference if it is a symmetric operator such as a symmetric hash-join. We observe that, when one child of a join is an EDB and the other one is a recursive IDB, the join pull order can significantly affect the number of intermediate result tuples. More precisely, we comparatively evaluate three execution strategies:

- `build EDB`: The join operator first consumes all data from one input and builds a hash-table in memory. It then streams the other input and probes the hash table. Note that it is only possible to fully consume the input that is not recursive.
- `pull IDB`: The join only consumes data from its EDB child if no data is available on the recursive IDB input.
- `pull EDB`: Opposite to `pull IDB`, the join only consumes its IDB child if no data is available on the EDB child.
- `pull alter`: The join pulls alternatively from the two children without favoring one over the other.

The presence and impact of the intermediate result tuples depend both on the execution strategy and the application. In GalaxyEvolution (Figure 2.2), the number of intermediate result tuples is the same independent of the execution strategy. However, for other applications, the number of intermediate tuples varies when the execution strategy changes. To better illustrate this point, we consider two additional applications.

Consider the connected components application shown in Figure 2.5, which computes the connected components in a graph. In this application, rule (1) initializes the connected components: each vertex starts as its own connected component with its identifier. Rules (2) and (3) recursively compute the connected components: For all combinations of facts that satisfy the bodies, the aggregate function `$Min(v)` keeps and propagates only the current minimal component ID v for each vertex y. The evaluation of `$Min(v)` is inter-twined with the discovery of new facts, where intermediate result tuples are generated until convergence. This is true in both the synchronous and asynchronous modes, but the number of intermediate result tuples varies when the join pull order changes in the asynchronous mode. Intuitively, if the join between `Edges` and the newly updated component values from `CC` favors the recursive input (`pull IDB` execution method), then it prioritizes the propagation of values closer to convergence, which ultimately reduces the number of intermediate result tuples.

We also consider another application from the bibliometrics domain. The application computes

```
CC(x,x)          :-   Edges(x,_)              (1)
CC(y,$Min(v))    :-   CC(x,v), Edges(x,y)     (2)
                 :-   CC(y,v)                 (3)
```

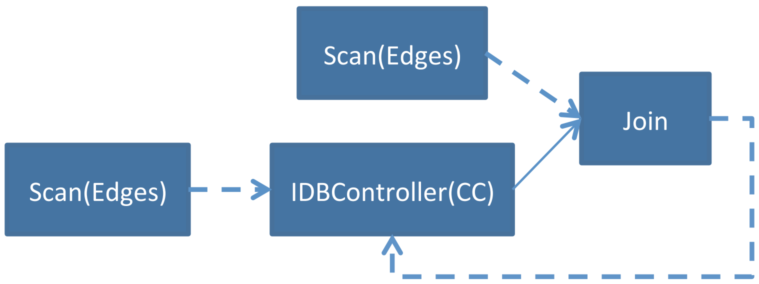


Figure 2.5: Datalog query (top) and recursive query plan (bottom) for connected components. The input EDB Edges(x,y) contains follower-followee edges.

the least common ancestor (LCA) for pairs of publications in a citations graph. An ancestor a of a paper p is any paper that is transitively cited by p, and the LCA a of two papers p_1 and p_2 is the least ancestor of both p_1 and p_2. Ancestor order is defined by the triple: (depth, year, paper_id). Figure 2.6 shows the Datalog and query plan for this application. The IDB `Ancestor` uses the aggregate function `$Min` to keep the length of the shortest path between two papers. In the synchronous mode, each iteration i only generates new pairs of papers with shortest path lengths equal to i. Once such a tuple is emitted, it will never be replaced by another tuple, which means there will be no unnecessary intermediate result tuples in this mode. In the asynchronous mode, however, a tuple of `Ancestor` may be replaced by another tuple with a smaller path length, which leads to a larger number of intermediate result tuples. The number of intermediate result tuples grows even larger in rule (3) with a self-join on `Ancestor`.

We evaluate the performance implications of these different execution alternatives on these three applications in Section 2.5.2.

2.3 Failure Handling

Several techniques exist to handle failures during the execution of iterative queries. The simplest approach is to restart the entire computation. The most well-known alternatives include data materialization at synchronization boundaries [55, 71, 221], checkpointing the state of the entire

```
Ancestor(b,a,1)                         :- Cite(b,a), b<seed                 (1)
Ancestor(p,a,$Min(depth+1))             :- Ancestor(p,b,depth),Cite(b,a)     (2)
LCA(p1,p2,$Min(greater(d1,d2),year,a)) :- Ancestor(p1,a,d1),Ancestor(p2,a,d2),
                                           Paper(a,year), p1<p2              (3)
```

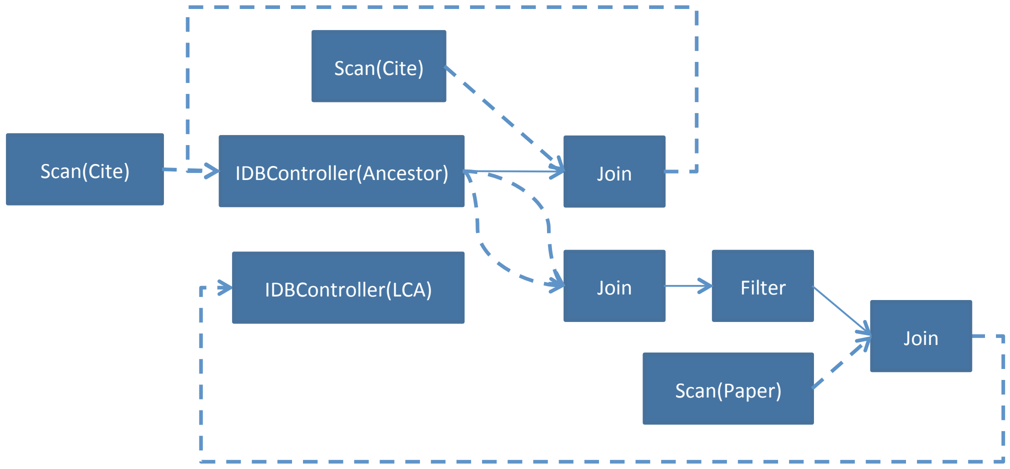


Figure 2.6: Datalog query (top) and recursive query plan (bottom) for LCA. The inputs are two EDBs: Paper(a,year) and Cite(b,a). The computation produces two outputs: Ancestor(b,a,depth) and LCA(p1,p2,depth,year,a).

computation either synchronously [152, 159] or asynchronously [146], and restarting using lineage tracking and periodic checkpoints [184, 225].

An important goal of our work is to develop synchronous and asynchronous iterative query processing methods that are simple to add to an existing shared-nothing engine. For this reason, we focus on failure handling methods that can be implemented by simply inserting failure-handling operators into query plans rather than modifying all operators (and queues) with the ability to checkpoint and recover state. We do not develop a new failure-handling method. Instead, we study fault-tolerance methods in the context of iterative query plans.

Similar to MapReduce [71] and Hadoop [221], we focus on fault-tolerance methods that buffer data on the producer side of data shuffling operators in the query plan. When a shuffle consumer worker fails, the shuffle producer resends the buffered data to a newly scheduled instance of the failed worker. Unlike MapReduce, but similar to the River system [47], the upstream backup methods developed for stream processing engines [115] and also used with shared-nothing database

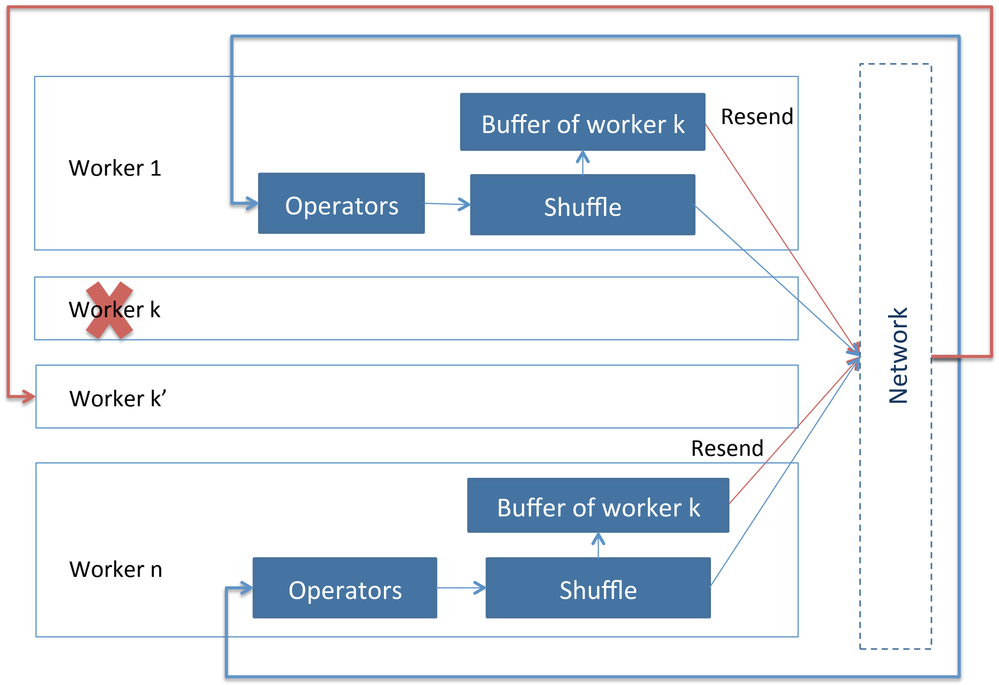


Figure 2.7: Fault-tolerance through data buffering in shuffle producer operators. When worker k fails, a new worker is re-scheduled and all other workers re-send their buffered data.

management systems [205], we buffer the data *in-memory and without blocking*. Figure 2.7 illustrates the approach. Buffers can spill to disk but we did not find that necessary in the applications that we used in the experiments.

For failure detection, we use simple heartbeat messages from the workers to the master, but we could also use more sophisticated cluster configuration methods [210].

During normal computation, each worker buffers its outgoing messages to other workers in memory in these shuffle operator buffers. If a worker fails, the master starts a new worker process and reschedules the failed query fragment on that process. The approach could also recover the failed fragments in parallel using multiple workers to speed-up recovery [184, 225]. The newly scheduled fragments process the iterative query from the beginning, while all other workers resend their buffered data.

The above failure handling methods are known. Our contribution is to study how amenable iterative computations are to optimizations that are possible for these buffer-based fault-tolerance methods. We study the following optimizations:

- `Append Buffer`: Buffer all data in FIFO queues with no optimization.
- `Aggregate Buffer`: The idea is equivalent to using a MapReduce combiner. In the case when the data being buffered is part of an IDB with aggregation, the data can be partially aggregated at the sender.
- `Prioritized Buffer`: Prior work [227] has shown that prioritizing the execution of specific tuples can speed-up convergence of iterative computations. For example, in the case of Connected Components, prioritizing tuples with the lowest component IDs can help to propagate these lower values faster. The idea of the prioritized buffer is to support such prioritization during failure recovery be re-ordering tuples in the buffer based on an application-specific priority function. For Connected Components, the sort order is increasing on component ID.

We empirically compare the overhead and recovery time of the above failure-handling methods in Section 2.5.3.

2.4 Implementation

We implement our approach in the Myria [23, 102] data management system. Myria's query execution layer, called MyriaX, is a shared-nothing distributed engine, where there is one master node and multiple worker nodes. As in HadoopDB [40], datasets ingested into Myria are sharded into PostgreSQL databases local to each node. MyriaX can read from other sources but we use PostgreSQL in our experiments. Once data is read out of PostgreSQL it is processed entirely in memory.

MyriaX is a relational engine. Query plans comprise relational algebra operators that are partitioned across workers. To distribute data across operator partitions, we use hash-partitioning and insert data shuffling operators to perform data re-distribution when necessary. Within each worker, query execution is pull-based: each operator produces output by pulling data from its children and returning it to its parent. Communication between workers is push-based: producer operators aggressively push data to consumers, with backpressure-based flow control used to balance the rates of data production and consumption while keeping the dataflow pipeline full. MyriaX processes tuples in batches to remove function call and network protocol overheads.

To justify our choice of the Myria engine for the implementation and evaluation of recursive query plans, we compare Myria's basic query execution performance to Spark [225], a state-of-the-art engine that includes support for synchronous iterative computations.

Figure 2.8 shows the results [3], [4]. Each bar shows the ratio of query execution time of Spark over Myria. Selection, aggregation and connected components are running on top of the full Twitter [133] dataset, which contains approximately 41 million vertices and 1.5 billion edges of the "follower, followee" relationships. For join, we use a subset of Twitter, which contains 60 thousand vertices and 1.5 million edges, because Spark could not produce results when a larger subset was used due to large memory usage. The join result has around 400 billion tuples. In all cases, Myria outperforms Spark with also a smaller variance in query execution times. These experiments illustrate that MyriaX achieves state of the art performance on standard queries, including iterative queries and is thus a good platform for the study of the performance differences between recursive query plan execution methods presented in this paper.

2.5 Evaluation

In this section, we evaluate the performance of our recursive query plans. The experiments address the following: (1) Does asynchronous query evaluation always lead to the fastest run times? (2) Do the variants of asynchronous evaluation (Section 2.2.3) matter? When does each variant lead to the fastest query run time and why? (3) Which fault-tolerance approach yields the best trade-off between run time overhead and failure recovery time?

We evaluate our techniques using three applications and datasets:

- Connected Components (Figure 2.5): We compute the connected components on a subset of the Twitter graph [133], which contains 21 million vertices and 776 million edges. To study how

[3] Results are generated in memory, not materialized to disk.

[4] Here we use round-robin partitioning for all datasets in Myria to make it a fair comparison with Spark and HDFS.

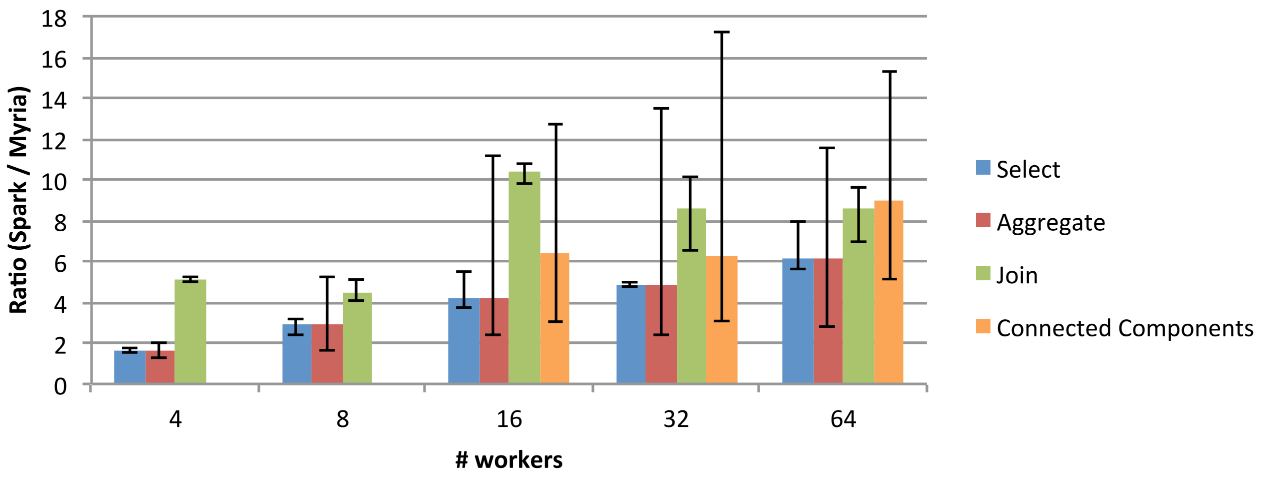


Figure 2.8: A comparison of Spark and MyriaX on four queries: Select (`R(x,y) :- Twitter(x,y), x < 5000000`)**, Aggregate** (`R(x,sum(y)) :- Twitter(x,y)`)**, Join** (`R(x,z) :- Twitter(x,y), Twitter(y,z)`)**, and Connected Components. MyriaX completes these queries** $1.5\times$ **to** $10\times$ **faster on average than Spark. For Connected Components, Spark (using GraphX) runs out of memory for small cluster sizes and large data.**

the graph degree distribution may affect results, we also compute the connected components for three synthetic graphs generated using Snap [138]. These graphs are power-law graphs obtained by varying the exponent of the power law distribution. Each graph has 21 million vertices and either 192 (dense), 81 (medium), or 20 (sparse) million edges.

- GalaxyEvolution (Figure 2.2): We compute galactic merger graphs on an astronomy simulation [27] that is 80GB in size with 27 timesteps.
- LCA (Figure 2.6): We determine the least common ancestor in a real bibliometrics dataset obtained from a UW collaborator containing 2 million papers and 8 million citations.

2.5.1 Experimental Method

We run all experiments using our Myria prototype implementation (Section 2.4) in a 16-node shared-nothing cluster interconnected by 10 Gbps Ethernet. Each machine has four Intel Xeon CPU E5-2430L 2.00GHz processors with 6 cores, 64GB DDR3 RAM and four 7200rpm hard drives. We vary cluster size using 8 or 16 machines with 1 to 4 worker processes each.

In each experiment, we measure the run time and resource consumption of each query while it executes until convergence. We report the query run time, total CPU time across all workers, total network I/O (number of tuples sent), and the *maximum* memory consumption for the entire query (number of tuples in operator states and buffers). All queries are executed five times, and we report the average values along with min/max values as error bars. Unless stated otherwise, we use round-robin partitioning for the base data (EDBs), hash-based partitioning for the operators in the query plans, present resource consumption results for the 32-worker cluster configuration (other sizes exhibit the same trends), and normalize the figures to the resource consumption of the best evaluation strategy to enable comparison across resource types.

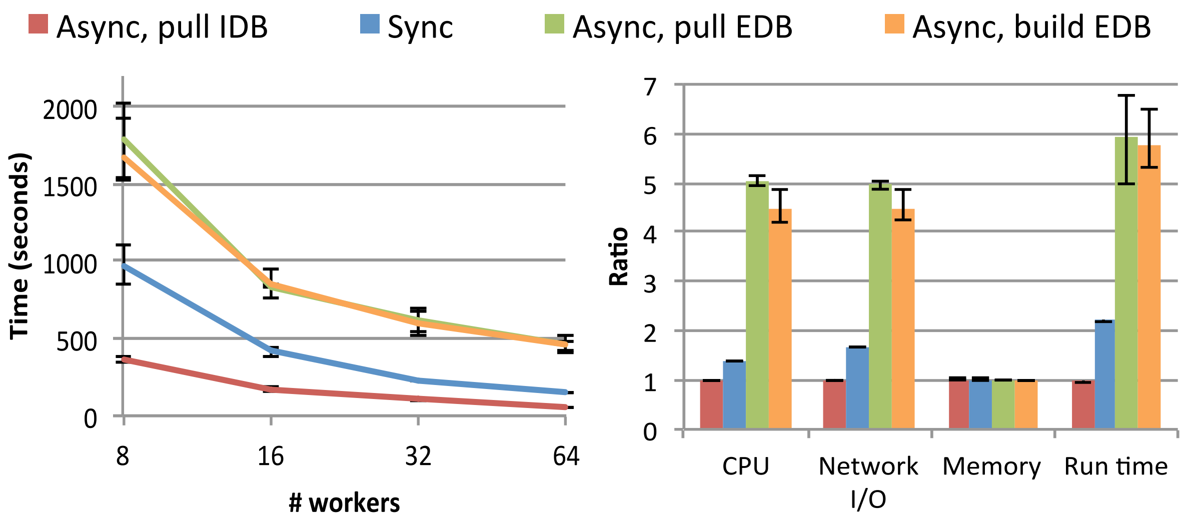
2.5.2 Execution Model

In this section, we evaluate the performance of the various execution models described in Section 2.2.3. In particular, we evaluate synchronous versus asynchronous execution strategies and, for asynchronous execution, compare how EDB-first, IDB-first, or balanced pull orders affect convergence.

Connected Components

Figure 2.9 shows the query run time and resource consumption results for Connected Components on Twitter: we find that using asynchronous execution and preferring new IDB results (`(async, pull IDB)`) yields both the fastest query run time and the lowest overall resource utilization. Synchronous iteration is about a factor of 2 slower. This result is expected, as the benefits of asynchronous execution for Connected Components have been widely reported [157, 179].

However, we surprisingly find that *synchronous iteration is significantly faster* than `(async, pull EDB)` and `(async, build EDB)`. Asynchronous models range from $2\times$ faster to $6\times$ slower than synchronous, depending on how they propagate data. For this query, pulling from the IDB as much as possible helps to propagate small component IDs faster across the network to `CC` on remote nodes. This helps reduce the amount of intermediate result tuples significantly and thus achieves



(a) Query run times of different cluster sizes and execution models

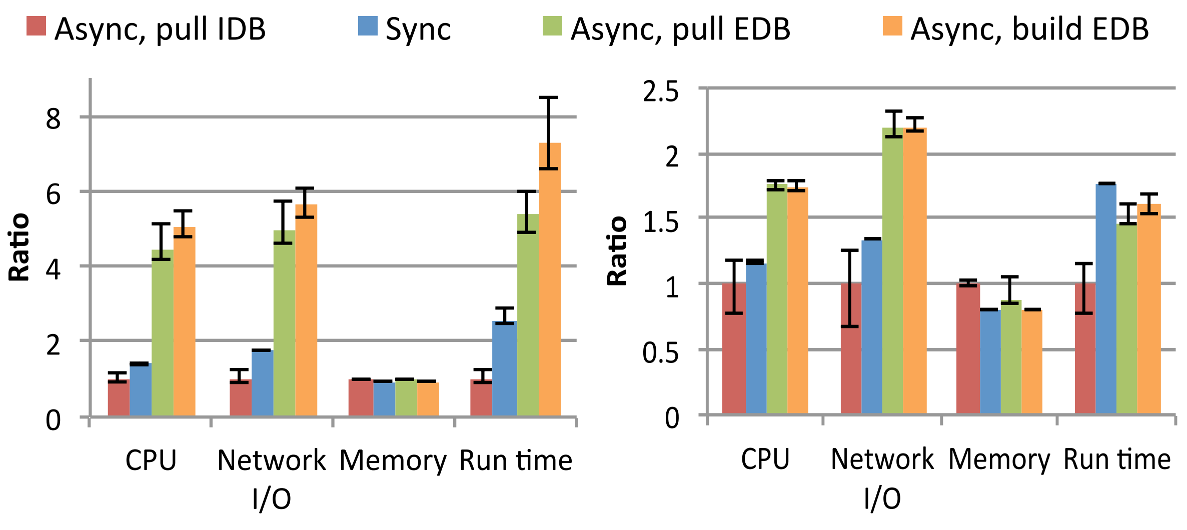
(b) Relative resource consumption of different execution models with 32 workers. `(Async, pull IDB)` serves as reference.

Figure 2.9: Connected Components: query run times and resource consumption.

faster convergence and lower resource consumption. In contrast, the strategies that prefer to load the EDB into memory generate many intermediate tuples that are later replaced; `build EDB` is slightly faster than `pull EDB` because it only builds a single hash table, saving some computation.

The synchronous model achieves a middle ground between the asynchronous strategies. It is slower than `(async, pull IDB)` because of the global barrier between each iteration step. However, it is able to aggregate away intermediate results at the barriers, and this reduction in redundant work dominates the EDB strategies. The sizes of intermediate results are immediately visible among all four techniques when considering the network I/O: `(async, pull IDB)` shuffles fewer than one quarter of the tuples of the other asynchronous methods.

Figure 2.10 shows the results on the synthetic datasets. We only show the resource consumption of the dense and the sparse datasets because the pattern of the medium dataset falls between them. The dense dataset yields similar results to the Twitter dataset. In contrast, for the sparse dataset, `sync` becomes the slowest strategy in terms of query run time. This is caused by the long tail of the sparse graph. However, for other types of resource consumption, `sync` still sits in between the two



(a) Dense (b) Sparse

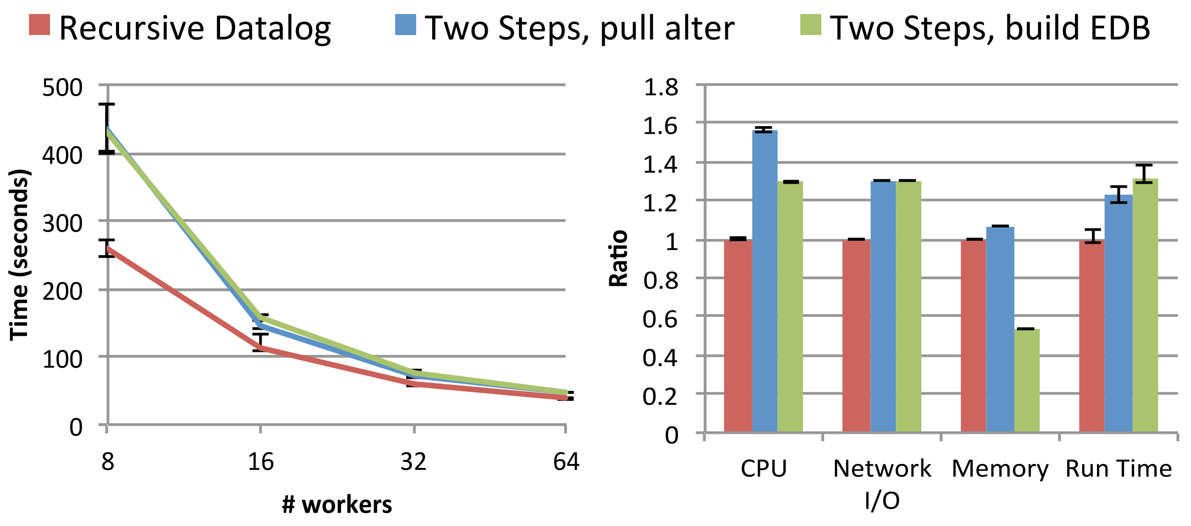
Figure 2.10: Connected Components on synthetic datasets: relative resource consumption of different execution models with 32 workers. `(Async, pull IDB)` **serves as reference.**

asynchronous strategies, which is similar to the Twitter dataset.

To our knowledge, even though Connected Components is such a well-studied problem, no prior report has illustrated the subtleties in how strongly the choice of execution strategy affects distributed system performance.

GalaxyEvolution

An important benefit of our approach is its focus on general-purpose Datalog programs as opposed to focusing only on processing graphs as in the case of SociaLite [179, 180] and several other engines [146, 212]. In GalaxyEvolution, the input data is a relation tracking particles through galaxies over time, and the goal is to compute the historical merger graph for a set of galaxies of interest at present day. As we described above, there are two ways to query the data. One approach Figure 2.1 first computes a full `Edges` relation for the galaxies in the simulation, in essence a full graph of galaxies, then extracts the sub-graphs reachable from the galaxies of interest. The alternate approach (Figure 2.2) represents the entire computation directly using recursive Datalog with our novel support for bag-monotonic aggregation.



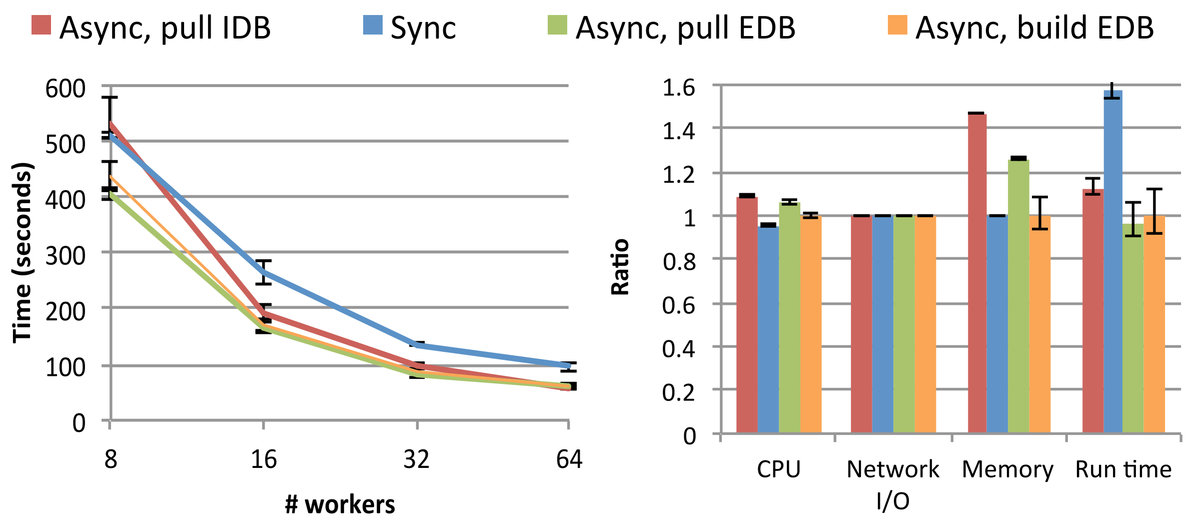
(a) Query run times of two-step and recursive Datalog.

(b) Relative resource consumption of two-step and recursive Datalog with 32 workers. Recursive Datalog serves as reference.

Figure 2.11: GalaxyEvolution: query run times and resource consumption, two-step versus recursive Datalog.

In this experiment, we randomly select one percent of the present day galaxies as the groups of interest. We compare our novel recursive Datalog plan (the choice of strategy does not matter, for reasons we discuss below) to the two-step approach using two execution models: `pull alter` for the join in step 1 or `build EDB` for building the EDB hash table first since they have different memory consumption on hash tables. As the results in Figure 2.11 show, using recursive Datalog leads to a 25% faster total run time and a lower resource utilization except for memory. The performance and network I/O gains are explained because we avoid the computation of unnecessary `Edges`. The higher memory utilization comes from having both recursive join operators active at the same time in one query rather than computing them one at a time in two separate steps.

Next, we focus on the recursive Datalog approach. Figure 2.12 shows the performance of different execution methods. To emphasize the differences between these models, we change the selectivity of the GalaxyEvolution query by using all galaxies in the groups of interest and also lowering the `thresh` to ensure that the bottleneck of the query is not the disk I/O.



(a) Query run times of different cluster sizes and execution models.

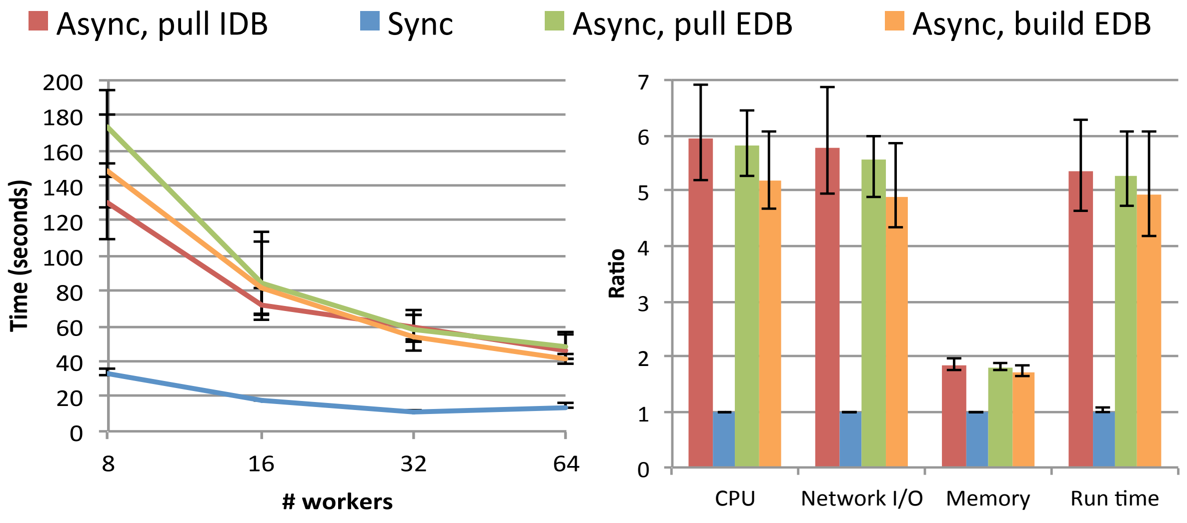
(b) Relative resource consumption of different execution models with 32 workers. `(Async, build EDB)` serves as reference.

Figure 2.12: GalaxyEvolution: query run times and resource consumption.

As we can see, `(async, pull EDB)` and `(async, build EDB)` yield the lowest run time. The latter does so with less memory because it never builds a hash table for the IDB input. These results are in contrast with Connected Components, where `(async, pull IDB)` is the most efficient execution model. The key reason is that there are no invalid intermediate results as GalaxyEvolution converges to a fixpoint; each step learns new facts about the next timestamp. Hence, prioritizing the IDB does not help to converge faster, and instead the symmetric joins employed by `pull IDB` and `pull EDB` spend time updating two hash tables, though `pull EDB` finishes one child and switches to a single-sided join earlier. The `sync` uses the same amount of resources as the best `async` methods but it is slower because of the global barrier.

Least Common Ancestor

Figure 2.13 shows the performance results for the LCA application. As the Datalog program is essentially stratified by depth, the synchronous execution model always finds ancestors at the lowest depth and has no unnecessary intermediate result tuples in the `Ancestor` relation. In contrast,



(a) Query run times of different cluster sizes and execution models.

(b) Relative resource consumption of different execution models with 32 workers. `(Async, pull IDB)` serves as reference.

Figure 2.13: LCA: query run times and resource consumption.

the asynchronous model generates many such intermediate result tuples in `Ancestor`, and even more such tuples are generated when `Ancestor` is joined with itself to compute the `LCA` relation. Because of the large number of intermediate result tuples, and their quadratic impact on result size, all `async` strategies yield much worse performance than `sync`. In general, the three `async` models have similar resource consumption numbers when varying the cluster size, although in this figure, `(async, build EDB)` slightly outperforms the other methods.

Summary

In summary, our experiments show the following trends:

- *Asynchronous query evaluation does not always lead to the fastest query run times.* For Connected Components `async` only works well combined with the right execution strategy (join pull order), and for stratified applications like LCA, asynchronous query evaluation performs unnecessary work that synchronous evaluation can avoid.
- *The variants of asynchronous evaluation (Section 2.2.3) have a big impact on query run time.*

Our study of Connected Components shows, for the first time, that variants can *significantly* affect performance. The system should favor propagating newly-generated IDB tuples only when it will not generate many intermediate result tuples; otherwise using single-sided joins with fewer hash tables saves computation.

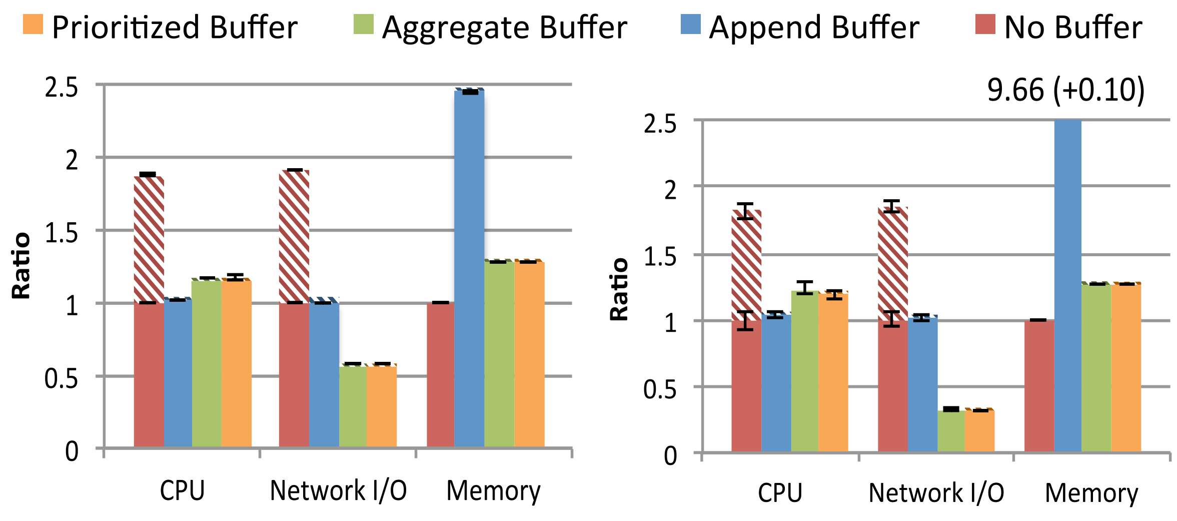
2.5.3 Failure Handling

In this subsection, we evaluate the failure-handling approaches described in Section 2.3 on the same three applications. For each query, we first evaluate the resource consumption overhead of fault-tolerance in the absence of failures. Then, we kill one worker during the query execution, and compare the total amount of resources used to process the query using either query restart (`No Buffer`) or one of the three buffer-based methods described in Section 2.3. We measure resource consumption instead of run time as it measures the total overhead on the cluster independent of how the recovery tasks are scheduled. We show the results when killing one worker approximately 70% of the time into the query execution. Similar patterns, though with somewhat different values, emerge when killing a worker earlier during the query execution.

We perform all experiments twice. First, we randomly partition EDBs. These EDBs must be shuffled during the query execution. Second, we hash-partition EDBs before the query execution such that they only need to be read locally. The difference between the two approaches lies only in the number of shuffle operators and in-memory buffers: when EDBs are hash-partitioned, shuffles after scans are not needed, which saves in-memory buffers. We find that all the trends are *identical* for both scenarios. The overheads are uniformly somewhat larger when an extra shuffle operator is added. We thus only show the results with the hash-partitioned EDBs.

Figure 2.14 shows the fault-tolerance overheads of the `(async, pull IDB)` or `(async, pull EDB)` execution methods. Each bar represents the ratio of resource utilization of one buffer type compared with execution without any failure handling. The filled portions at the bottom show the ratios in the absence of failures, while the portions with diagonal stripes on top show the additional overhead to recover from a failure.

In the absence of failures, maintaining buffers in shuffle operators adds overhead. Basic Append



(a) Pull from IDB (b) Pull from EDB

Figure 2.14: Connected Components: relative resource consumption of different buffers with 32 workers. Filled bars: no failure, patterned bars: overhead to recover from a failure. No Buffer, no failure serves as reference. In (b), the memory consumption of the Append Buffer reaches 9.66 **without failures and** 9.76 **with failure. It is truncated in the figure.**

buffers add significant memory overhead, especially for `(async, pull EDB)` as it generates more intermediate tuples. The Aggregate and Prioritized buffers dramatically cut memory and network I/O overheads while only minimally increasing CPU overheads. These two methods have even lower network I/O than no buffering due to the data aggregation they perform before shuffling.

In the case of failure, all three buffer-based methods incurred *negligible* overhead due to failure recovery. Because this overhead is negligible, the extra work of sorting tuples in the buffer to prioritize the execution is unnecessary. As a result, the Aggregate Buffer delivers the best trade-off in terms of total resource consumption with or without failures.

Figure 2.15 shows the results for the GalaxyEvolution application. Aggregate Buffer and Prioritized Buffer are not applicable to this application since the only aggregate function is `$Count`. Similar to Connected Components, using Append Buffer consumes more memory during normal execution, but saves CPU time and network bandwidth in case of failures. Once again, the total resource consumption is nearly identical for an execution without failure or one with one failed

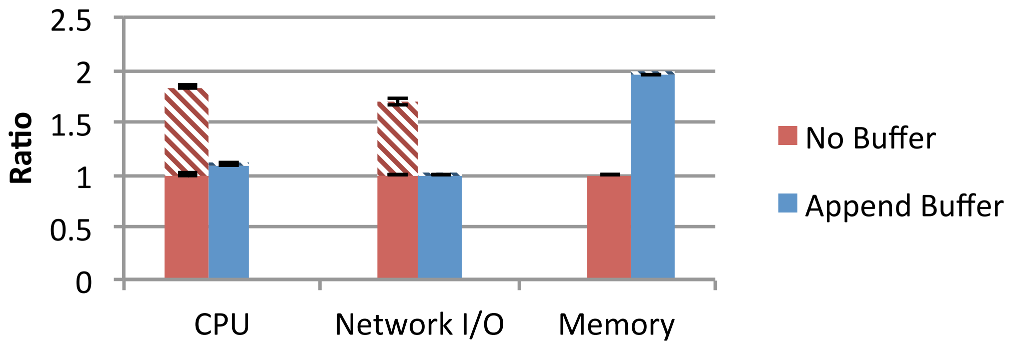


Figure 2.15: GalaxyEvolution: relative resource consumption of different buffers with 32 workers. Filled bars: no failure, patterned bars: overhead to recover from a failure. No Buffer, no failure serves as reference.

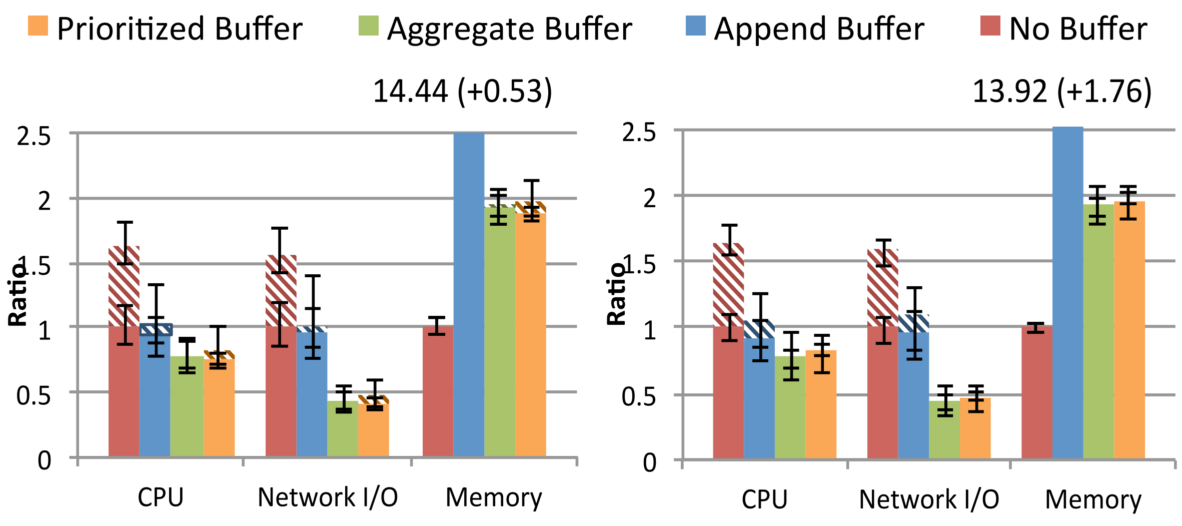
worker.

Finally, Figure 2.16 shows the results for LCA. The trends are the same as for Connected Components. An interesting effect is that the early aggregation in the recovery buffers reduces total CPU consumption even in the absence of failures. Importantly, total resource utilization in the presence of a failure is, once again, nearly identical to the resource utilization without failure for all three buffer-based methods showing that prioritization during recovery is not necessary.

In summary, a lightweight buffer-based method for failure handling yields only a small increase in CPU utilization in the absence of failures, yet can recover from failures with negligible added CPU cost. The memory overhead of data buffering can be large but extending the buffers with early aggregates dramatically cuts these costs, which stay within 2X in all three applications tested. Interestingly, the failure-handling extensions reduce network I/O even without failures.

2.6 Summary

In this chapter, we developed an approach for large-scale iterative data analytics that combines the benefits of many existing systems. Users express their analysis in recursive Datalog with aggregation, which simplifies the expression of analytics from a variety of application domains. The system executes the analysis using parallel query plans that require only small extensions to an existing shared-nothing engine yet deliver the full power of incremental synchronous and



(a) Pull from IDB (b) Pull from EDB

Figure 2.16: LCA: relative resource consumption of different buffers with 32 workers. Filled bars: no failure, patterned bars: overhead to recover from a failure. No Buffer, no failure serves as reference. The memory consumption of the Append Buffer reaches 14.44 **without failures and** 14.97 **with failure in (a), and** 13.92 **without failures and** 15.68 **with failure in (b). They are truncated in the figure.**

asynchronous query evaluation even for query plans with multiple recursive IDBs. Finally, we empirically evaluate when different variants of query execution and failure handling methods deliver the fastest query run time for different applications. We find that no single method outperforms others. The key factor is the amount of intermediate tuples generated during query evaluation.

Algorithm 1 Translate a Datalog program into an asynchronous recursive query plan

```
1.  function GenerAtePlan($P$)
2.      Input: Datalog program $P$
3.      $R \leftarrow$ Set of all rules in $P$
4.      $X \leftarrow$ Set of all IDBs in $P$
5.      $Y \leftarrow$ Set of all EDBs in $P$
6.      Instantiate an TerminationController $E$
7.      for ($x \in X$) do
8.          Instantiate an IDBController $C_x$ with empty inputs
9.          Connect control output of $C_x$ to input of $E$
10.     for ($y \in Y$) do
11.         Instantiate a Scan $S_y$
12.     for ($r \in R$ with $h \in X$ as head) do
13.         Translate the body of $r$ into a relational query plan $Q_r$
14.         for ($x \in X$ that appears in the body of $r$) do
15.             Connect output of $C_x$ to $Q_r$ as input
16.         for ($y \in Y$ that appears in the body of $r$) do
17.             Connect output of $S_y$ to $Q_r$ as input
18.         if (body of $r$ contains only EDBs) then
19.             Union the output of $Q_r$ with the initial input of $C_h$
20.         else
21.             Union the output of $Q_r$ with the recursive input of $C_h$
22.     for ($x \in X$) do
23.         if (x has nothing on its initial input) then
24.             Instantiate and connect a Scan($\emptyset$) to its initial input
25.         if (x has nothing on its recursive input) then
26.             Instantiate and connect a Scan($\emptyset$) to its recursive input
```

Chapter 3

ELASTICMEM: ELASTIC MEMORY MANAGEMENT FOR CLOUD DATA ANALYTICS

In this chapter, we focus on the problem of elastic memory management for cloud data analytics. The work presented in this chapter appeared at USENIX ATC'17 [215]. As we introduced in Section 1.3, large-scale analytics needs large amounts of computing resources, and many big data systems are designed to run in clusters or cloud environments to leverage abundant resources. Resources in such environments are often shared by multiple systems, users, and applications, and a resource manager is often used for scheduling and resource allocation. Modern resource managers rely on containers (*e.g.*, YARN [207]), which have hard resource limits to both protect and isolate applications. As an example, Figure 3.1 illustrates the interaction between a resource manager and containers. The resource manager launches containers with resource limits and schedules applications inside those containers.

To use containers, users or service providers need to estimate the resource needs of an application before its execution. However, estimating resource usage, especially memory usage for large-scale analytical queries, is a hard problem, and inaccurate memory usage estimates can harm query performance in multiple ways. If the estimate is too high, cluster resources may be under-utilized. If too low, the system must either spill data to disk, which degrades performance, or fail with an out-of-memory error, wasting the resources already consumed by the query. This challenge exists in systems with manual memory management, such as those written in C/C++ [129, 146], in Java-based systems that use byte arrays [32], and in systems that rely on automatic memory management provided by runtimes, such as Java [6, 213, 221, 225] and the .NET Common Language Runtime (CLR) [159]. The situation becomes more complicated when garbage collection (GC) is used for automatic memory management since GC activities add another layer of unpredictability

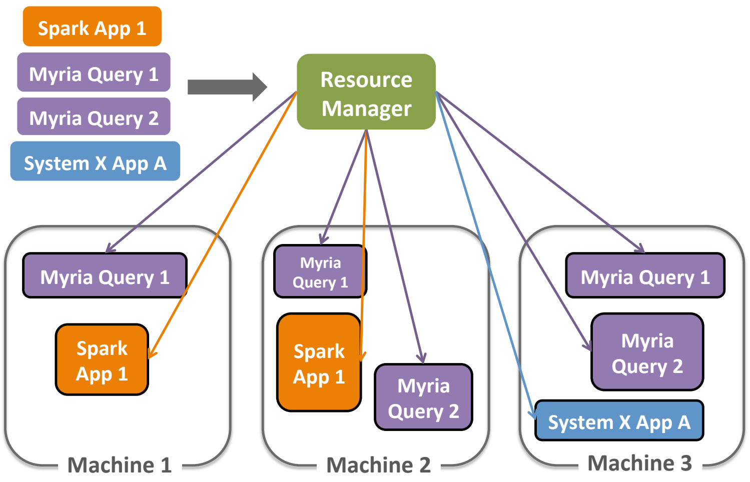


Figure 3.1: A resource manager schedules multiple applications from multiple systems (Spark [225], Myria [213], and System X) in a shared cluster. An application may have multiple processes across multiple machines. The resource manager schedules applications by putting them in containers with resource limits.

to query performance. Even if resource estimates are sufficient to complete the query, garbage collection in some cases can significantly slow down query execution. As a concrect example, we demonstrate (Section 3.1) how changing the maximum heap size of Java-based systems significantly impacts query execution time.

To avoid memory usage estimation before execution forced by hard container memory limit, one possible solution is to make container memory limit *elastic*, *i.e.*, changing the limit dynamically during runtime. However, elastic container memory management presents difficulties. First, most systems do not support it. For Java-based systems, the maximum heap size of a Java virtual machine (JVM) stays constant during its lifetime. For C/C++-based systems, such as Impala [129], limiting process resources is usually done using Linux utilities (such as `cgroups`), which do not expose functionality to change resource limits at runtime. For systems that run in CLR [159], the problem is the opposite: control on heap size cannot be specified, so it can grow arbitrarily up to the total physical memory. Second, in order to elastically and dynamically allocate memory to data analytics applications, we must understand how extra memory can prevent failures and speed up these applications. We need models of GC benefits and overheads. Finally, we need an algorithm that uses the models to orchestrate memory allocation across multiple data analytics applications.

In this chapter, we present ElasticMem [214, 215], an approach for dynamically allocating memory across multiple applications in a shared cluster. Our solution is to avoid the rigid design of container resource hard limits by proposing an elastic way to manage memory, where container memory limits can be changed dynamically during runtime. Our focus is on analytical applications, in particular *relational algebra* queries on large data, and Java-based systems. Since memory management in Java containers (e.g., YARN [207]) is determined by JVMs internally, we focus on how and when to change the memory layouts of JVMs. Specifically, our contributions are the following:

- We show how to modify the JVM to enable dynamic changes to an application's heap layout for elastic management of its memory utilization (Section 3.2.1).
- Our key contribution is an algorithm for elastically managing memory across multiple applications in a big data analytics system to achieve an overall optimization goal (Section 3.2.2). In this paper, we present scenarios where each query runs in one JVM and multiple queries run in one machine, but our approach can be extended to a multi-machine setting.
- In support of elastic memory management, we develop a machine-learning based technique for predicting the heap state and GC overhead for a relational query and whether it is expected to run out of memory (Section 3.2.3) based on operator statistics. Since the common approach for implementing relational operators in memory, such as joins and aggregates, is to use hash tables [94], we build models that use hash table statistics as input.

We evaluate our elastic memory management techniques using TPC-H queries [29] on Myria [102, 213], a shared-nothing data analytics system, against containers with fixed memory limits. In our experiments, our approach outperforms static allocation: It reduces the number of query failures; it reduces query times by up to 30%, GC times by up to 80%, and overall resource utilization (Section 3.3).

3.1 Performance Impact of Automatic Memory Management

Many big data analytics systems today, including Spark [225], Flink [5], Hadoop [221], Giraph [6], and Myria [213], are written in programming languages with automatic memory management, specifically Java. Garbage collection associated with automatic memory management is known to cause performance variations that are hard to control: The GC policy, although customizable by the programmer to some extent, is controlled by the runtime internally. Depending on the policy and heap state, the time and frequency of GCs may vary significantly and, as we later show in this section, may significantly impact query performance.

Over the past decade, there have been several JVM implementations with various GC algorithms. However, most of the contemporary ones share the concept of generations [38]. With this design, the heap space is partitioned into multiple generations for storing objects with different ages. Figure 3.2 illustrates the internal state of a JVM heap with two generations. Initial memory allocation requests always go to the young generation. When it fills up, a GC is triggered to clean up dead objects. There are different types of GCs as shown in Figure 3.2. In a young collection, live objects in the young generation are promoted to the old generation. In a full collection, dead objects are cleaned from both generations in addition to promotions. The type of collection to trigger depends on whether a promotion failure, *i.e.*, insufficient space for promoting objects from the young generation, is expected to occur or actually occurs. In this paper, we use OpenJDK as the reference JVM implementation. We focus on the common class of GC algorithms that use a young and old generation, and leave extensions to other languages and GC algorithms to future work.

We show a concrete example of how GC can impact query execution by executing a self-join query on a synthetic dataset containing ten million tuples with two `int` columns, on three systems: Myria, Spark 1.1 and Spark 2.0, using one process on one machine with default GC collectors (`-XX:+UseParallelGC`). Figure 3.3 shows the query execution times with different heap-size limits. Each data point is the average of five trials with error bars showing the minimum and maximum values. For both Myria and Spark 2.0, when the heap is large, the query time converges to approximately 35 seconds, which is the pure query time with almost no GC. When we shrink the

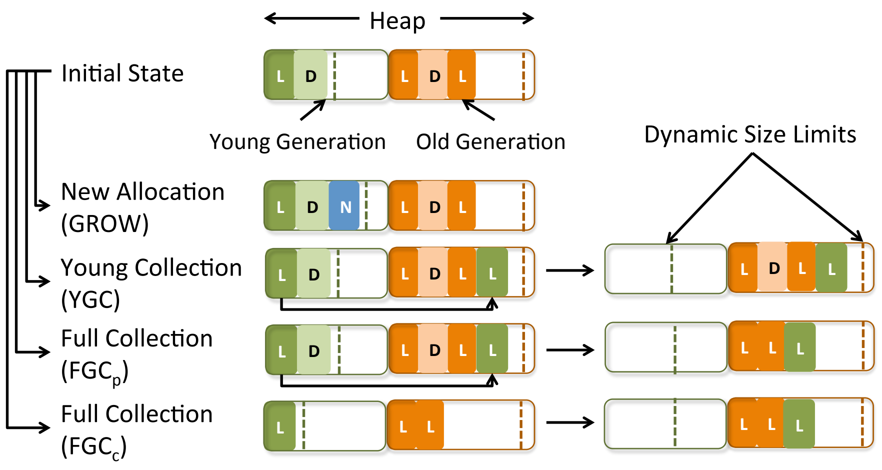


Figure 3.2: Internal heap states of a JVM before and after actions of new object allocation, young generation collection, and full collections, starting from an initial state. Dark blocks: (L)ive objects, light blocks: (D)ead objects, blue blocks: (N)ew objects. Dashed lines: generation size limits that can be changed in real time by our approach. We describe `FGC`$_p$ and `FGC`$_c$ in Section 3.2.2.

heap size, however, the run times increase moderately due to more GC time. For Myria, the run time increases from 35 seconds to 55 seconds when the heap size goes from 16 GB to 3 GB, and further increases drastically to 141 seconds when the heap size shrinks from 3 GB to 2 GB. Eventually, Myria fails with an out-of-memory error when the limit is less than 2 GB. Similarly, the query time for Spark 1.1 has a steep increase from 86 to 466 seconds when the heap size changes from 5 to 4 GB, and the query fails when the heap size is less than 4 GB. Spark 2.0 follows a similar trend as Myria, but does not fail even with only 500 MB of memory because it is able to spill data to disk when memory is insufficient. As a result, however, its execution time increases to 127 seconds.

3.2 Elastic Memory Allocation

In this section, we present our approach, called *ElasticMem*, for elastic memory allocation. ElasticMem comprises three key components. First, ElasticMem needs JVMs that can change memory limits dynamically, and we describe how we modify OpenJDK to enable this feature in Section 3.2.1. Second, the heart of ElasticMem is a memory manager that dynamically allocates memory across multiple queries (Section 3.2.2). Finally, to drive the manager's allocation decisions,

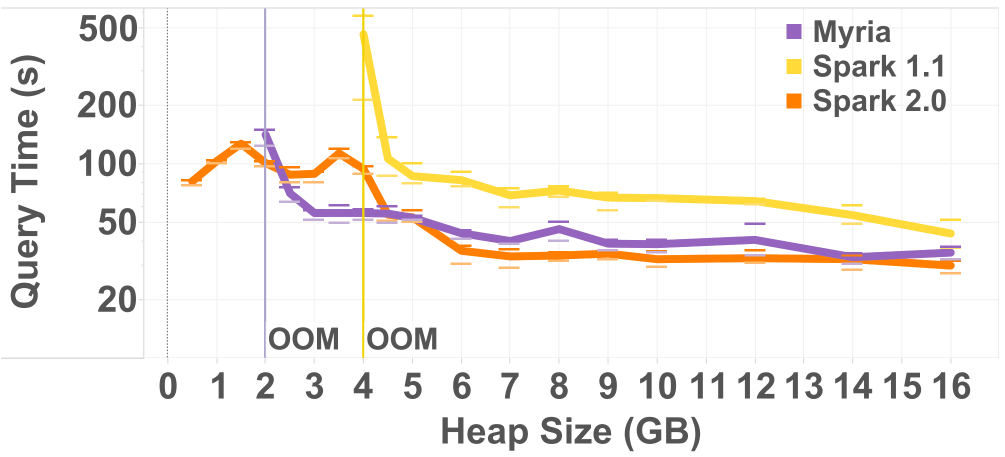


Figure 3.3: Impact of GC on query execution time in Myria, Spark 1.1, and Spark 2.0. The y-axis uses a log scale.

ElasticMem uses models that predict the heap state and the GC costs (*i.e.*, impact on run time) and benefits (*i.e.*, expected freed memory) at any point during query execution (Section 3.2.3).

3.2.1 Implementing Dynamic Heap Adjustment in a JVM

OpenJDK manages an application's memory as follows: First, the user specifies the maximum heap size of a JVM process before launching it. The JVM then asks the operating system to reserve the heap space and divides the space into generations based on its internal size policy as in Figure 3.2. During program execution, if a memory allocation request cannot be satisfied due to insufficient memory, the JVM may trigger GCs to release some memory. If not much memory is released after spending a large amount of time on GC, the JVM throws an `OutOfMemory` error. The maximum heap size stays constant during a JVM's lifetime. It cannot be increased even if an `OutOfMemory` is thrown while more memory is available on the machine, or decreased if heap space is underutilized.

This rigid design, however, is unnecessary. For operating systems that support overcommitting memory, a logical address space does not physically occupy any memory until it is used. This property, together with 64-bit address spaces, allow us to reserve and commit a large address space when launching a JVM. The actual memory limits on heap spaces, such as generations, can be modified later during runtime.

We modify the source code of OpenJDK to implement this feature. We change the JVM to reserve and commit a continuous address space of a specified maximum heap size (`-Xmx`) when it launches. The initial size limit of each generation is set according to the JVM's internal policy. We make the maximum heap size large enough such that the per-generation limits are sufficiently large to become irrelevant. Additionally, we add our new dynamic size limits to both the young and old generation of a JVM p, denoted with $y_{limit}(p)$ and $o_{limit}(p)$ respectively. Our memory manager changes these limits at runtime. We set their initial values to reasonably small numbers (*e.g.*, 1 GB) and prevent each generation from using more memory than its dynamic limit.

To interact with the JVM, we add a socket-based API through which the JVM receives instructions such as requests for the current heap state, memory limit adjustments, or GC triggers. We disable the JVM's internal GC policies to let our memory manager control when and which GCs to happen. We modify GC implementations to always release recycled memory to the OS. If more memory is needed but unavailable given the current limits, we let the JVM pause until more memory is available. We implement our changes on top of OpenJDK 7u85's default heap implementation (`ParallelScavengeHeap`), which contains approximately 1000 lines of code.

3.2.2 Dynamic Memory Allocation

The main component of ElasticMem is a memory manager. It monitors concurrently executing queries and alters their JVMs' memory utilizations by performing actions on the JVMs, such as triggering a GC or killing the JVM. Each action has a value, and the objective is to maximize the sum of all action values. A value is a combination of several factors, including whether the action kills a JVM, causes a JVM to pause, or how efficiently it enables the JVM to acquire memory: *i.e.*, the ratio of time spent over space acquired (from the OS or recovered through a GC).

The manager makes decisions according to two pieces of information: the JVM heap states and the estimated values of performing actions on the JVMs. Because predicting these values far into the future carries significant uncertainty, and because our changes to the JVM enable us to adjust memory limits without any overhead, we develop a dynamic memory manager. The manager makes decisions adaptively at each timestep t for some small period $[t, t + \delta_t]$. At t, the manager gathers runtime statistics from each JVM and performs actions on it. Queries then execute for time δ_t. Their states change and the manager makes another round of decisions at $t + \delta_t$. We describe our allocation algorithms in this section, starting with a more precise problem statement.

Problem Statement

We start with a single-node and a one-process-per-query scenario. As introduced in Chapter 3, each JVM is a container that executes a single query (or query partition). We model query execution as the process of accommodating the memory growth of the corresponding JVM. For a period $[t, t + \delta_t]$, the memory usage of a JVM may grow by some amount. We can perform various actions to the JVM to affect its memory utilization: allocate enough memory for the expected growth, trigger a GC, which may require extra memory in the short term but free up memory in the longer term, kill the JVM to release all its memory, or do nothing, which may stall a JVM if it cannot grow its memory utilization as needed.

Consider a single physical machine with a total amount of memory M. A set of N JVMs $\{p_1, \ldots, p_N\}$ is running on it, each has used some space in both the young and the old generation. At the current timestep t, we need to allocate M across the N JVMs, such that the total memory used does not exceed M, while minimizing a global objective function.

The memory that must be allocated to a JVM is entirely determined by the action that the manager selects. For example, to perform a young generation GC, the old generation needs to have enough space to accommodate the promoted young generation live objects. The manager must increase the memory limit for the old generation to accommodate the added space requirement. We denote with $y_{cap}(p_i, a_i)$ and $o_{cap}(p_i, a_i)$, the minimal amount of memory that must be allocated to the young and old generation of JVM p_i, if the manager chooses action a_i. These values refer to the new required totals and not increments.

Each action has a value that contributes to the global objective function. We denote the value of action a_i on p_i with $value(p_i, a_i)$. The objective function is thus:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} value(p_i, a_i), a_i \in Actions, \\ \text{subject to} \quad & \sum_{i=1}^{N} (y_{cap}(p_i, a_i) + o_{cap}(p_i, a_i)) \leq M, \end{aligned}$$

where $Actions$ is the set of possible actions. In our approach, the $value(p_i, a_i)$ is a structure with multiple fields. We describe its internal structure and how to sum and compare values in Section 3.2.2 below.

The above definition can be extended to a shared-nothing cluster scenario by letting the manager make decisions independently for each machine.

Runtime Metrics

Several runtime metrics are needed to compute the value and the space requirements of actions. Some are reported by the JVM while others are estimated by the manager:

Metrics reported by the JVM: For a JVM p at timestep t, $y_{limit}(p, t)$ and $o_{limit}(p, t)$ are the current memory limits of the young and old generation. The manager sets those limits at the previous timestep. However, only some of the space in each generation is used at t, and the JVM reports the used sizes as $y_{used}(p, t)$ and $o_{used}(p, t)$.

Metrics estimated by the manager: Besides the above metrics, we also need to estimate some values that are not directly available. First, the space used in the young and old generation of a JVM is further divided into live and dead objects. The manager estimates the total size of those objects, which we denote with $\hat{y}_{live}(p, t)$, $\hat{y}_{dead}(p, t)$, $\hat{o}_{live}(p, t)$ and $\hat{o}_{dead}(p, t)$. We use $\hat{x}$ to indicate that a value x is estimated by the manager. Second, the manager needs to estimate p's heap growth, $\hat{grw}(p, t)$, before the next timestep, where $\hat{grw}(p, t) = \hat{y}_{used}(p, t+\delta_t) - y_{used}(p, t)$. Finally, to model the impact of a GC, the manager needs to know how much memory a GC will free, and how much time it will take. Since the target of a GC is the set of all objects in the generation(s) undergoing the GC, we use $y_{obj}(p, t)$ to denote the set of all the objects in the young generation and similarly $o_{obj}(p, t)$ for the old generation.[1] $\hat{gc}_y(y_{obj}(p, t))$ and $\hat{gc}_o(o_{obj}(p, t))$ are then the estimated times for

[1] $y_{obj}(p, t)$ is the union of all the live and dead objects in the young generation of p at t, similarly to $o_{obj}(p, t)$.

Value	Meaning
$y_{limit}(p, t)$	Size limit of the young gen
$o_{limit}(p, t)$	Size limit of the old gen
$y_{used}(p, t)$	Total used space in the young gen
$o_{used}(p, t)$	Total used space in the old gen
$\hat{y}_{live}(p, t)$	Total size of live objects in the young gen
$\hat{o}_{live}(p, t)$	Total size of live objects in the old gen
$\hat{y}_{dead}(p, t)$	Total size of dead objects in the young gen
$\hat{o}_{dead}(p, t)$	Total size of dead objects in the old gen
$g\hat{r}w(p, t)$	Estimated heap growth until next timestep
$\hat{gc}_y(y_{obj}(p, t))$	Time to perform a young collection
$\hat{gc}_o(o_{obj}(p, t))$	Time to perform an old collection

Table 3.1: Runtime metrics reported by JVM p or estimated by the manager at timestep t. $\hat{x}$ indicates that x is estimated. "gen" is short for generation.

a young and an old GC. We describe how the manager estimates these metrics in Section 3.2.3.

Table 3.1 summarizes the notation. Since t is the only used timestep, we omit t and only use p as the argument in the rest of the paper when the context is clear.

Space of Possible Actions

There are four types of actions that the manager can choose for each JVM: allowing the JVM to grow by asking the operating system for more memory, reducing the memory assigned to the JVM by performing a garbage collection and recycling space,[2] pausing the JVM if it cannot either grow or recycle enough memory, or as a last resort, killing a JVM to release its entire memory. The manager performs an action for every JVM at each timestep. An action a on a JVM p has value, $value(p, a)$, with a minimum amount of memory needed for p's young and old generations, ($y_{cap}(p, a)$ and $o_{cap}(p, a)$). We denote the time to perform a on p with $time(p, a)$, and the size of

[2]The recycled memory is always reclaimed by the OS.

the newly available space made by a with $space(p, a)$. The *cost* of an action is the amount of time needed to acquire a given amount of space, or $\frac{time(p,a)}{space(p,a)}$. The manager uses this ratio to compare and choose actions.

The detailed set of $Actions$ is as follows:

- `GROW`: Let the JVM grow to continue query execution. In order to reserve space for the growth, the manager must allocate $y_{cap}(p, \texttt{GROW}) = y_{used}(p) + \hat{grw}(p)$ to the young generation and $o_{cap}(p, \texttt{GROW}) = \hat{y}_{live}(p) + o_{used}(p)$ to the old generation. We reserve extra space in the old generation for prospective promotions to preserve the possibilities of having all types of GCs in the future. The cost is the time it takes to request and access the new space, which depends on the size of the space change given by: $y_{cap}(p, \texttt{GROW}) + o_{cap}(p, \texttt{GROW}) - y_{limit}(p) - o_{limit}(p)$. Under normal circumstances, this will be the commonly selected action until space becomes tight and JVMs must start garbage collection or must pause before being able to grow again.
- `YGC`: Trigger a young generation GC. The JVM needs at least the current used space, $y_{used}(p)$, for the young generation, and $\hat{y}_{live}(p) + o_{used}(p)$ for the old generation to avoid a promotion failure. The cost is the GC time $\hat{gc}_y(y_{obj}(p))$, and we expect memory of size $\hat{y}_{dead}(p)$ to be recycled.
- $\texttt{FGC}_\texttt{p}$: Trigger a full GC by first performing a young generation collection to promote live objects to the old generation then performing a GC on the old generation. Similar to `YGC`, we need at least $y_{used}(p)$ and $\hat{y}_{live}(p) + o_{used}(p)$ for the young and old generations respectively. The cost is the GC time $\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p))$ and the space to be recycled is $\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$.
- $\texttt{FGC}_\texttt{c}$: Trigger a full GC by first performing a GC on the whole heap, then trying to promote young generation live objects if possible, without changing the total heap size. Free space from the young generation after the first GC gets shifted to the old generation to make space for copying. Different from $\texttt{FGC}_\texttt{p}$, we only need $y_{used}(p)$ and $o_{used}(p)$ for the young and old generation since the promotion is not mandatory. However, more GC time is needed since the full collection is now performed on both generations instead of only the old generation. We assume that the time grows proportionally to the size of live objects and use $\hat{gc}_y(y_{obj}(p)) + \hat{gc}_o(o_{obj}(p)) * (\hat{y}_{live}(p) + \hat{o}_{live}(p))/\hat{o}_{live(p)}$ as the GC time estimate. The memory to be recycled is also $\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$.

- `NOOP`: Do nothing to the JVM, keep the current limits $y_{limit}(p)$ and $o_{limit}(p)$. As a consequence, the JVM is expected to pause since it cannot either grow or recycle enough memory by doing garbage collection.
- `KILL`: Kill the JVM immediately. As a consequence, the query running in this JVM will fail.

$\texttt{FGC}_\texttt{p}$, which promotes first, is the default behavior in OpenJDK. However, the promotion may fail if the old generation does not have enough free space to absorb young generation live objects, and when it happens, JVM spends much time on copying the live objects back, so that the young generation remains the same as it was before the promotion. In other words, triggering $\texttt{FGC}_\texttt{p}$ while expecting a promotion failure is not cost effective. However, when memory is scarce, the manager may not be able to allocate extra space to avoid the promotion failure. In this case, we need a GC which can still recycle space without increasing the limits. We solve this problem by implementing another full GC procedure, $\texttt{FGC}_\texttt{c}$: We first collect both generations, then shift young generation free space to the old generation to keep the total heap limit unchanged. A young GC is then performed if there is enough space for promoting.

Table 3.2 summarizes the properties of all actions. $os(m)$ denotes the time to access new memory of size m. We obtain its value by running a calibration program since this value changes for different systems and settings. Figure 3.2 illustrates the effect of all the actions except for `NOOP` and `KILL`, which have the obvious effects.

We define the value of an action with three attributes, where only one of them is set to a non-zero value. For `NOOP` and `KILL`, we set the corresponding attributes to 1. For other actions, we use their cost, or time/space efficiency, as the value: *i.e.*, how much time the action needs per unit of space that it makes available. Then for an action a on a VM p, its value $value(p, a)$ is defined as:

$$\begin{cases} value(p,a).cost = \dfrac{time(p,a)}{space(p,a)}, & \text{for} \quad \texttt{GROW}, \texttt{YGC}, \\ & \qquad\quad \texttt{FGC}_\texttt{p}, \texttt{FGC}_\texttt{c}, \\ value(p,a).\texttt{NOOP} = 1, & \text{for} \quad \texttt{NOOP}, \\ value(p,a).\texttt{KILL} = 1, & \text{for} \quad \texttt{KILL}, \end{cases}$$

With the above definition, our manager can favor actions by comparing these three attributes in a

Action a	$\mathbf{y_{cap}(p,a)}$	$\mathbf{o_{cap}(p,a)}$	$\mathbf{space(p,a)}$	$\mathbf{time(p,a)}$
GROW	$y_{used}(p)$ $+\hat{grw}(p)$	$\hat{y}_{live}(p) + o_{used}(p)$	$y_{cap}(p,a) + o_{cap}(p,a)$ $-y_{limit}(p) - o_{limit}(p)$	$os(space(p,a))$
YGC	$y_{used}(p)$	$\hat{y}_{live}(p) + o_{used}(p)$	$\hat{y}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))$
FGC$_{\mathrm{p}}$	$y_{used}(p)$	$\hat{y}_{live}(p) + o_{used}(p)$	$\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))$ $+\hat{gc}_o(o_{obj}(p))$
FGC$_{\mathrm{c}}$	$y_{used}(p)$	$o_{used}(p)$	$\hat{y}_{dead}(p) + \hat{o}_{dead}(p)$	$\hat{gc}_y(y_{obj}(p))$ $+\hat{gc}_o(o_{obj}(p)) * r,$ $r = \frac{\hat{y}_{live}(p)+\hat{o}_{live}(p)}{\hat{o}_{live}(p)}$
NOOP	$y_{limit}(p)$	$o_{limit}(p)$		
KILL	0	0		

Table 3.2: Per-generation size limit requirements, sizes of created space, and time taken for each action a in $Actions$ on JVM p at the current timestep. $os(m)$ is the time to access memory of size m. Other symbols are defined in Table 3.1.

certain order, as we describe in Section 3.2.2.

Memory Allocation Algorithm

Next, we discuss the allocation algorithm, which allocates memory to the JVMs by performing actions on them at each timestep. We model the problem as a 0-1 knapsack problem. The capacity of the knapsack is the total amount of memory, and the items are actions performed on JVMs. Each action has a value and a minimum space requirement as described in Table 3.2. The goal is to maximize the total item value in the knapsack without exceeding its capacity.

The 0-1 knapsack problem is known to be NP-complete with a pseudo-polynomial dynamic programming solution [66]. Let $opt_{N,M}$ denote the value of the best scheme for allocating memory of size M to the first N JVMs, $p_1 \cdots p_N$. If a JVM p_i is undergoing a GC, the manager skips it to wait for the GC to complete. Otherwise, it derives $opt_{i,j}$ by enumerating possible actions on p_i and picking the one that leads to the largest value for $opt_{i,j}$. We define the sum of two values as the sum of their three attributes, then the state transition function is defined as:

$$opt_{i,j} = \begin{cases} opt_{i,j} & \text{if } opt_{i,j} > opt_{i-1,j-m} + v, \\ opt_{i-1,j-m} + v & \text{otherwise}, \end{cases}$$

$$\text{where } v = value(p_i, a), a \in Actions, i \in [1, N], j \in [0, M].$$

To choose between two values, we first check which one has a lower value for attribute `KILL`, then fewer `NOOP`s to reduce pausing time, then a smaller time/space ratio. The one with fewer `KILL`, then fewer `NOOP`, then smaller time/space ratio, has a *higher* value. To be precise, given two values a and b, we define $a > b$ as:

```
bool operator>(const Value& a,
               const Value& b) {
  if a.KILL < b.KILL return true
  if a.NOOP < b.NOOP return true
  if a.cost < b.cost return true
  return false
}
```

The complexity of the dynamic programming is $O(N * M)$, where N is the number of JVMs and M is the total amount of memory. On modern servers, M can be large if the memory-size units are fine-grained, which would prevent the manager from making fast decisions. At the same time, allocating memory at fine granularity is unnecessary. To enable fast memory-allocation decisions, we define U as the unit of memory allocation, and any allocation is represented as a multiple of U. We discuss two ways of setting U: as a constant or as a dynamically computed variable based on the current heap state, and evaluate their impact on performance in Section 3.3.

Algorithm 2 and Algorithm 3 show the detailed allocation algorithms. Function `ALLOCATE` allocates memory of size M across the list of JVMs, P, at the current timestep, and it returns the best allocation scheme, act^{best}, which is a vector of actions for each $p \in P$. The algorithm works as follows: First, we find all the JVMs that are not undergoing a GC as $P - P_{\texttt{INGC}}$ to compute their actions. Because the algorithm allocates memory as increments of U, but $y_{limit}(p)$

and $o_{limit}(p)$ of a JVM p at the current timestep may not be increments of U when U is a dynamic variable, we do not include `NOOP` in Algorithm 3. Instead, we consider all the combinations of $P - P_{\texttt{INGC}}$ as potential $P_{\texttt{NOOP}}$ (line 4) and use $P' = P - (P_{\texttt{INGC}} \cup P_{\texttt{NOOP}})$ to denote the remaining JVMs. The remaining memory to be allocated is of size M' (line 7). We then apply Algorithm 3 on P' and memory of size M' ($= K$ units of size U). Function `KNAPSACK` returns the best solution with its value. The generation size limits and value of an action on a JVM are computed as in Table 3.2. The size limits are aligned to increments of U by function $align(size, U)$ defined as: $align(size, U) = ceiling(size/U)$. For GC actions, we defined a constant $mingcsave$ to avoid GCs that only recycle a negligible amount of space. We derive act from the transition actions $trans$ and return them together with the value. They are then merged with $P_{\texttt{NOOP}}$ and $P_{\texttt{INGC}}$ to get the final allocation. We maintain the best allocation and its value across all the powersets. In the end, if the best allocation only contains `NOOP` actions, we pick some JVMs to kill to make progress. In this work, we pick the query that occupies the largest amount of memory and kill all its JVMs, and we leave other strategies as future work.

3.2.3 Estimating Runtime Values

The last piece of ElasticMem is the models that estimate JVM values that are necessary for memory allocation decisions yet not directly available as indicated in Table 3.1.

Heap Growth

To allocate memory to a JVM for the next timestep, the memory manager needs to estimate its memory growth. Different approaches are possible. In this paper, we adopt a simple approach. To estimate the heap growth of JVM p at timestep t, $\hat{grw}(p, t)$, the manager maintains the maximum change in the young generation's usage during the past b timesteps. To be precise, we define: $\hat{grw}(p, t) = \max |y_{used}(p, t') - y_{used}(p, t' - \delta_t)|, t' \in [t - b * \delta_t, t]$. In our experiments, we set $b = 3$ empirically. We show in Section 3.3 that this value yields good performance.

Algorithm 2 The scheduling algorithm: allocates memory of size M across the list of JVMs P, returns the allocation scheme.

```
function ALLOCATE($P, M$)
    $value^{best} = act^{best} = None$
    $P_{\texttt{INGC}}$ = $\{p \in P, p$ is undergoing a GC$\}$
    for $P_{\texttt{NOOP}} \in$ power set of $P - P_{\texttt{INGC}}$ do
        $act_p = \texttt{NOOP}, p \in P_{\texttt{NOOP}}$
        $P' = P - (P_{\texttt{INGC}} \cup P_{\texttt{NOOP}})$
        $M' = M - \sum_{p \in P_{\texttt{INGC}} \cup P_{\texttt{NOOP}}} (y_{limit}(p) + o_{limit}(p))$
        Compute $U$, let $K = M'/U$
        $act', value' = Knapsack(P', K, U)$
        $act_p = act'_p, p \in P'$
        $value.cost = value'.cost$, $value.\texttt{KILL} = value'.\texttt{KILL}$
        $value.\texttt{NOOP} =$ size of $P_{\texttt{NOOP}}$
        if $value > value^{best}$ then
            $value^{best} = value$, $act^{best} = act$
    if $act^{best}$ contains only NOOP then
        Pick $P_{kill} \subseteq P$, let $act_p^{best} = \texttt{KILL}, p \in P_{kill}$
    return $act^{best}$
```

GC Time and Space Saving

The GC time and space saving depend primarily on the number and total size of the live and dead objects in the collected region. Unfortunately, getting such detailed statistics is expensive, as we need to traverse the object reference graph similarly as in a GC. Paying such a cost for each JVM at every prediction defeats the purpose of reducing GC costs in the first place.

We observe, however, that a query operator's data structures and their update patterns determine the state of live and dead objects, which determines GC times and the amount of reclaimable memory. Our approach is thus to monitor the state of major data structures in query operators,

Algorithm 3 The knapsack problem: given the list of JVMs P and K memory units of size U, returns the best allocation and its value.

```
1: function KNAPSACK($P, K, U$)
2:     $N$ = size of $P$
3:     $opt_{0,j} = 0, j \in [0, K]$
4:     for $i \leftarrow 1, N$ do
5:         for $j \leftarrow 0, K$ do
6:             for $a \in [\texttt{GROW}, \texttt{YGC}, \texttt{FGC}_\texttt{c}, \texttt{FGC}_\texttt{p}, \texttt{KILL}]$ do
7:                 if $a \in [\texttt{YGC}, \texttt{FGC}_\texttt{c}, \texttt{FGC}_\texttt{p}]$ and
8:                 $space(p_i, a) < mingcsave$ then continue
9:                 $y_{unit} = align(y_{cap}(p_i, a), U)$
10:                $o_{unit} = align(o_{cap}(p_i, a), U)$
11:                if $opt_{i-1,j-y_{unit}-o_{unit}}$ is valid then
12:                    $v = opt_{i-1,j-y_{unit}-o_{unit}} + value(p_i, a)$
13:                    if $v > opt_{i,j}$ then
14:                        $opt_{i,j} = v, trans_{i,j} = (a, y_{unit} + o_{unit})$
15:    Derive $act_p$ of each $p \in P$ from $opt_{N,K}$ and $trans_{N,K}$
16:    return $act, opt_{N,K}$
```

collect statistics from them as features, and use these features to build models. While there are many operators in a big data system, most keep their state in a small set of data structures, for example, hash tables. So instead of changing the operators, we wrap data structures with the functionality to report statistics, and instrument them during query execution to get per-data structure statistics. There are many large data structures, but in data analytics systems, the most commonly used ones by operators with large in-memory state, such as join and aggregate, are hash tables. In this paper, we focus on the hash table data structure. To get predictions for the whole query, we first build models for one hash table, then compute the sum of per-hash-table predictions as the prediction for the whole query. Our approach, however, can easily be extended to other data structures and

Feature	Meaning
nt	Total # of processed tuples
ntd	Delta # of processed tuples since the last GC
nk	Total # of distinct keys in the hash table
nkd	Delta # of distinct keys since the last GC
num_{long}	# of `long` columns
num_{str}	# of `String` columns
sum_{str}	Avg. sum of lengths of all `String` columns

Table 3.3: Features collected from a hash table.

operators.

Table 3.3 lists the statistics that we collect for a hash table. A hash table stores tuples consist of columns. A tuple has a key defined by some columns and a value formed by the remaining columns. We collect the number of tuples and keys in a hash table in both generations (both the total and the delta since the previous GC), since new objects are put in the young generation only until a GC. These features are nt, ntd, nk and nkd. The schema also affects memory consumption. In particular, primitive types, such as `long`, are stored internally using primitive arrays (e.g. `long[]`) in many systems that optimize memory consumption. However, data structures with Java object types, such as `String`, cannot be handled in the same way, as their representations have large overhead. So we treat them separately by introducing features for primitive types (num_{long}) and `String` types (num_{str} and sum_{str}). The overhead of getting these values from hash tables is negligible. We then build machine learning models to predict the GC times and the total size of live and dead objects as specified in Table 3.1.

To build models, our first approach to collect training examples is to randomly trigger GCs during execution to collect statistics. The models built from them, however, yielded poor predictions for test points that happen to fall in regions with insufficient training data. As a second approach, we collected training data using a coarse-grained multidimensional grid with one dimension per feature. The examples were uniformly distributed throughout the feature space but they all had the same small set of distinct feature values, the values from the grid. As a result, predictions were excellent for values on the grid but poor otherwise. Using a fine-grained grid, however, is too expensive since the feature space has eight dimensions. For example, if we divide each dimension in four, the total number of grid points is $(4+1)^7 = 78,125$. Assuming that collecting one data point requires 30 seconds, we need $78,125/2/60 \approx 651$ machine hours. Our final approach is thus to combine the previous two: We first collect data using a coarse-grained grid to ensure uniform coverage of the entire feature space, then for each grid cell, we introduce some diversity by collecting two randomly selected data points inside of it. The union of the grid and the random points is the training set. To collect a data point for a hash table, we run a query with only that hash table and a synthetically generated dataset as the input. This approach enables us to precisely control the feature values when we trigger a GC. We then can use any off-the-shelf approach to build a regression model. In our implementation, we use the M5P model [167, 217] from Weka [99] since it gives us the most accurate predictions overall. We evaluate our models in Section 3.3.2.

3.3 Evaluation

We evaluate the performance of our memory manager and the accuracy of our models. We perform all experiments on Amazon EC2 using `r3.4xlarge` instances. We do not set swap space to avoid performance degradation due to virtual memory swapping. We execute TPC-H queries [29] on Myria [213], a shared-nothing data management and analytics system written in Java. The TPC-H queries are written in MyriaL, which is Myria's declarative query language, and they are publicly available at [30]. We modify or omit several queries because MyriaL does not support some language features, such as nulls and `ORDER BY`. The final set consists of 17 TPC-H queries: *Q1-Q6*, *Q8-Q12*, and *Q14-19*. To experiment with a broad range of query memory consumption, we execute each query on two databases with scale factors one and two.

3.3.1 Scheduling

We first compare our elastic manager (*Elastic*) against the original JVM with fixed maximum heap size (*Original*). For Original, we assume that each running JVM gets an equal share of the total memory. We pick 4 memory-intensive TPC-H queries, Q4, Q9, Q18, and Q19, and execute each on two databases, which leads to a total of 8 queries. In all experiments, we execute these 8 queries on one EC2 instance together with our memory manager. All data points are averages of five trials, and we report the minimal and maximal values as floating error bars. Each run of the allocation algorithm takes about 0.15 seconds.

We empirically set the constant $mingcsave$ from Algorithm 3 to 30 MB. The value of the function $os(m)$ is obtained by running a calibration program, which asks the operating system for memory of size m using `mmap` and accesses it using variable assignments. We take the system time as $os(m)$. For `r3.4xlarge`, we get $os(m) = 0.35s * \frac{m}{1\text{ GB}}$. We set the interval between timesteps, δ_t, to 0.5 seconds except in Section 3.3.1, where we compare different values of δ_t. In order to avoid query hanging due to frequent GCs that do not recycle much memory, we kill a query after 8 minutes if it is still running. Based on our observation, 8 minutes is long enough for any query to complete with a reasonable amount of memory.

One extreme of Original is serial execution where queries are executed one at a time, while the other extreme is to execute all queries simultaneously. The former approach requires the least amount of memory for all queries to complete but takes longer time, while the latter finishes all queries the fastest when memory is sufficient, however may fail more queries when memory is scarce. We vary the degree of parallelism (*DOP*) for Original to compare these alternatives. To make it fair for Elastic, we also introduce a variant of Elastic, which allows executions to be delayed by resubmitting killed queries serially after all queries either complete or get killed. We call this variant *Elastic-Resubmit*. To avoid livelocks, we only resubmit each killed query once, and each resubmitted query runs only by itself. We leave resubmitting multiple queries simultaneously as future work.

Another important parameter is the size of the memory increment unit U. The value of U can

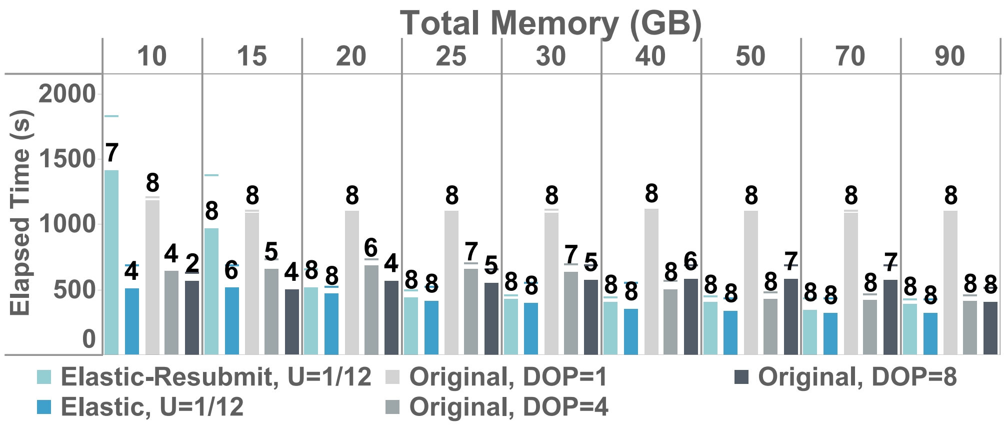


Figure 3.4: Average elapsed times and # of completed queries (labeled on top of each bar).

be either fixed or derived in real time. We test fixed sizes of 100 MB, 500 MB, and 1000 MB, and variable sizes as 1/8, 1/12, and 1/16 of the total free space at the current timestep.

Scheduling Simultaneous Queries

First, we submit all queries at the same time. Figure 3.4 shows the elapsed times, together with the numbers of completed queries while varying the total memory size. The elapsed times are the times for all queries to complete. In this figure, we use U=1/12 as the representative of our elastic manager because it provides the best overall performance across all experiments. We further discuss the performance of different values of U in Figure 3.5. When memory is abundant ($\geq$ 20 GB), both Elastic managers yield more completed queries and also shorter elapsed times than all the three Original variants. When memory is scarce ($\leq$ 15 GB) and only suffices to execute one query at a time, for 15 GB, Elastic-Resubmit is able to complete all queries with less time than Original, DOP=1. For 10 GB, it only misses one query with a slightly longer time comparing to DOP=1. Based on our observation, the query failed because our manager needs to allocate memory as increments of U, however U is not sufficiently fine-grained. The overhead of elapsed time is due to the elastic method striving to accommodate all queries together before degrading to serial execution. As a proof of concept, we calculate the in-memory sizes of dominant large hash tables of the 8 queries and find that the sum of them is about 14 GB. This experiment shows the advantage of

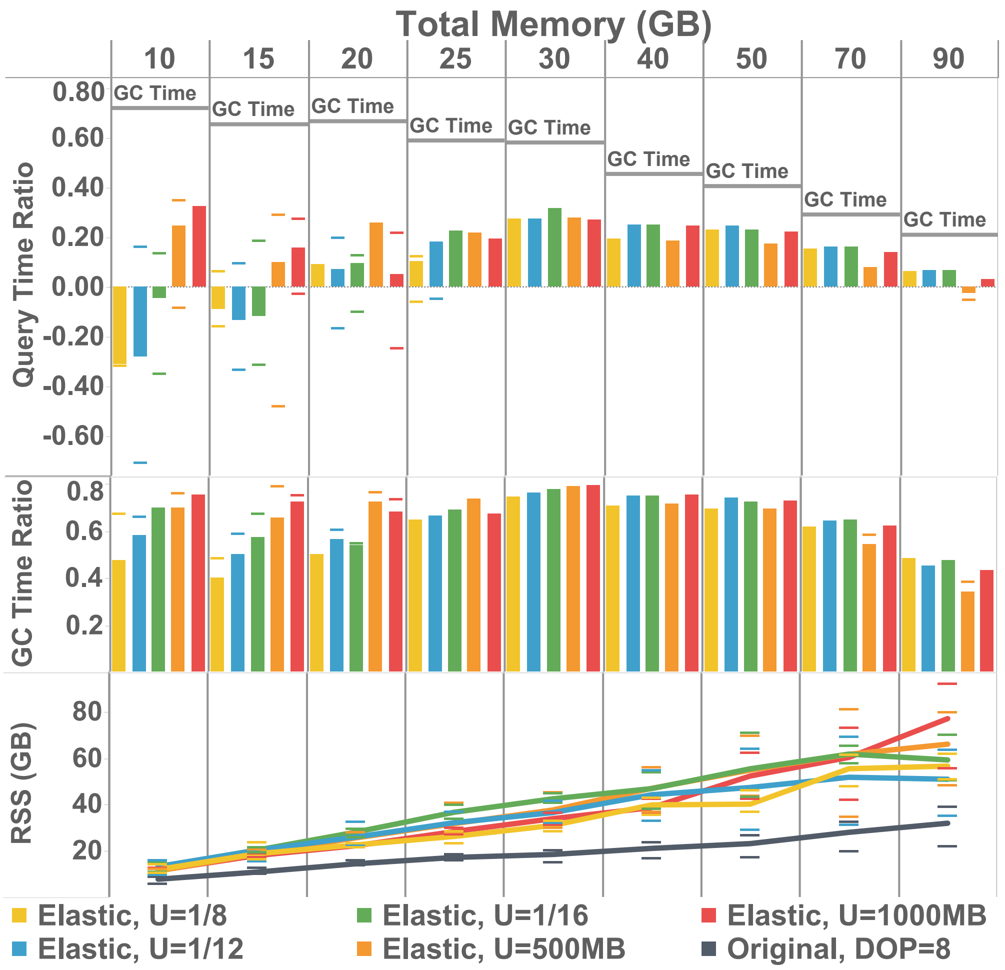


Figure 3.5: Relative total query time improvement ratio (top) and GC time improvement ratio (middle) for Elastic over Original, DOP=8, and resident set size, RSS (bottom).

using the elastic manager: it automatically adjusts the degree of parallelism, enabling the system to get high-performance while avoiding out-of-memory failures when possible.

In Figure 3.5, we further drill down on the performance of different variants of our approach. We seek to determine which variant yields the greatest performance improvement compared with non-elastic memory management. Because the elapsed times of Original, DOP=1 are significantly longer than the other two variants, we use Original, DOP=8 as the baseline in this experiment, which also brings fair comparison with our approach. We measure performance in terms of total query execution time, which is the sum of the per-query execution times, and total GC time, the sum of the GC times of all queries. Figure 3.5 shows the relative improvement percentages in total query execution time and GC time of Elastic over Original, DOP=8, for different values of U, and also the actual physical memory usage (resident set size, RSS).[3] Higher bars indicate greater improvements. When memory is scarce ($\leq$ 15 GB), Elastic with variable values of U (1/8, 1/12 and 1/16) takes longer to execute each query because it strives to finish more queries than Original, DOP=8, as shown previously in Figure 3.4. When memory is abundant ($\geq$ 20 GB), for any of the values of U, Elastic outperforms Original, DOP=8 on both total query time and GC time. The percentage improvements are between 10% and 30% for query time and 40% to 80% for GC time. We observe that it is caused by Original, DOP=8 triggering GCs that do not recycle much space especially in late stages for large queries but being unable to shift memory quota from small queries, while Elastic can dynamically allocate memory across all queries. The improvement ratios of query time decrease after 70 GB because GC time takes a less portion of query time when memory is abundant. To show the maximum improvement that we can achieve by reducing GC time to zero, we also show the ratios of total GC time to query time in the top subfigure as a reference. Finally, the bottom subfigure shows that our elastic manager is also able to utilize a larger fraction of available physical memory to save on GC time and query time. Importantly, all values of U, especially the three variable ones, yield similar performance indicating that careful tuning is not required.

Scheduling Queries with Delays

To better simulate a real cluster, instead of issuing all the queries at the same time, we submit the above 8 queries with delays. Each query is submitted 30 seconds later than the previous one. Figure 3.6 shows the elapsed times and the numbers of completed queries. The patterns are similar to the experiment above with no delay (Figure 3.4), but also different as Elastic can finish the same number of queries with less time when memory is scarce (10 GB), and always beats all variants of Original in terms of both query completion and elapsed time. This is due to the memory flexibility that ElasticMem has: the number of simultaneously running queries is lower when delay

[3] We define the improvement percentage as $(x - y)/x$, where x is the value of Original and y is the value of Elastic.

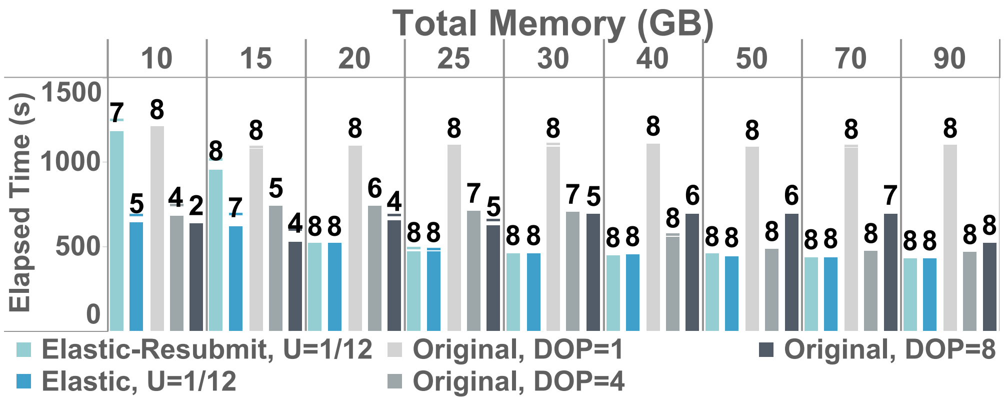


Figure 3.6: Average elapsed times and # of completed queries (labeled on top of each bar) with 30 seconds delay.

is introduced, so Elastic is able to finish more queries faster, while Original stays the same.

Timestep Interval

Finally, we evaluate the sensitivity of the approach to different values of δ_t varying from 0.1, 0.5, or 1 second for U=500MB and U=1/12. We find that when memory is scarce, 0.5 seconds slightly outperforms others by completing more queries with less time, although in general the three δ_ts yield similar performance, which indicates that the approach is not sensitive to small differences when using variable sizes of U and thus careful tuning is not necessary. We omit details due to space constraints.

3.3.2 GC Models

An important component of ElasticMem is its models that predict the GC time and the space that will be freed (Section 3.2.3). We evaluate its models in this section. We limit the training space to 12 million tuples and 12 million keys for a hash table, with the schema varying from 1 to 7 `long` columns and 0 to 8 `String` columns with a total of 0 to 96 characters. This training space is large enough to fit all hash tables from TPC-H queries. As described in Section 3.2.3, we collect approximately 1080 grid points and 1082 random points together as the training set. We also collect a test set of 7696 data points by randomly triggering GC for the 17 TPC-H queries on both

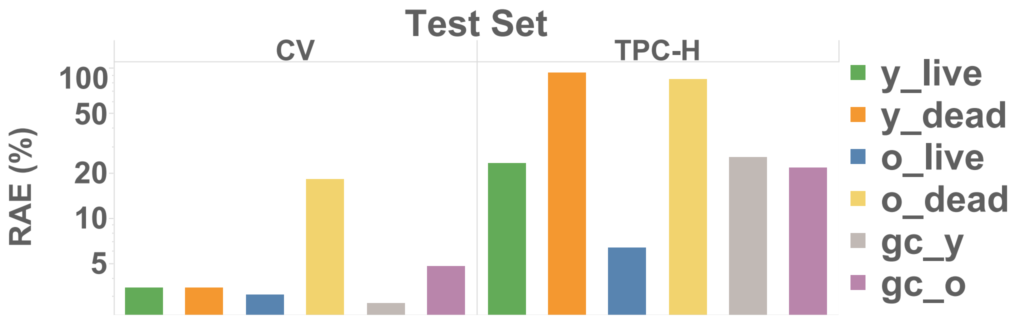


Figure 3.7: GC Model accuracies on 10-fold cross validation and random TPC-H test set.

databases.

We set the JVM to use one thread for GC (`-XX:ParallelGCThreads=1`) because we observe that the JVM is not always able to distribute work evenly across multiple GC threads. We do not use thread-local buffers (`-XX:-UseTLAB`). We let the JVM always sweep live objects to the beginning of the old generation after each collection (`-XX:MarkSweepAlwaysCompactCount=1`) instead of every few collections to reduce GC cost variance. Among several models available in Weka [99], we pick the M5P model with default settings for its overall accuracy. M5P is a decision tree where leaves are linear regressions [167, 217]. We use *relative absolute error* (RAE) to measure the prediction accuracies.[4]

Figure 3.7 shows the results for both doing 10-fold cross validation on the training set and testing on the random TPC-H test set. For cross validation, the predictions yield RAEs below 5% for every value except o_{dead}. For testing, both y_{dead} and o_{dead} cannot be predicted well, while all others have RAEs lower than 25%. This is because that the size of dead objects is not strongly correlated with the objects in data structures. Fortunately, the fact that the sum of dead and live objects is the total used size gives us a way to avoid predicting y_{dead} and o_{dead}. Instead, we let $\hat{y}_{dead} = y_{used} - \hat{y}_{live}$ and $\hat{o}_{dead} = o_{used} - \hat{o}_{live}$, where y_{used} and o_{used} can be obtained precisely. Overall, the prediction error rates are low and, as we showed in Section 3.3.1, suffice to achieve good memory allocation decisions.

[4]The RAE of a list of predictions P_i and corresponding real values R_i is defined as: $\sum_{i=1}^{n} |P_i - R_i| / \sum_{i=1}^{n} |\overline{R} - R_i|$.

3.4 Summary

We presented ElasticMem, an approach for the automatic and elastic memory management for big data analytics applications running in shared-nothing clusters. Our approach includes a technique to dynamically change JVM memory limits, an approach to model memory usage and garbage collection cost during query execution, and a memory manager that performs actions on JVMs to reduce total failures and run times. We evaluated our approach in Myria and showed that our approach outperformed static memory allocation both on query failures and execution times.

Chapter 4

Deluceva: DELTA-BASED NEURAL NETWORK INFERENCE FOR FAST VIDEO ANALYTICS

As we introduced in Section 1.4, modern data analytics involves not only traditional business data, but also more diverse data types such as images and videos. Additionally, recent advances in deep learning have made this type of image/video processing the mainstream approach for vision analytics.

Given the increasing availability of high-qualify, large-scale image datasets [15, 24, 131, 142] and high-performance computing devices, a large set of neural network models that perform various vision tasks on individual images have recently been developed. Common tasks include object classification [106, 119, 189, 198, 199], object tracking [79, 92, 160, 195], and object detection [68, 114, 144, 171, 172]. Some models are already superior to humans in both accuracy and speed [174]. As a result, many video analysis approaches [105, 124, 125, 171] apply image models directly to each video frame and then further process per-frame results. This type of approaches is easy to develop as any out-of-the-box image model can be applied directly to a video. However, the cost of running image models for each frame is high and continues to grow as models become more complex and inputs become larger. For example, representative object detection models run at speeds from less than one frame [172] to 90 frames [171] per second on high-end GPUs with low-resolution inputs (*e.g.*, 300×300). Higher resolution images and larger models are needed for more accurate and complex analysis [114], such as for detecting small obstacles in autonomous driving [67] as many accidents are caused by road debris or hazardous cargo [26, 31].

To improve efficiency, we develop a new approach to enabling more efficient analytics on real-world video streams. We focus on deep learning analytics and use object detection as concrete workloads. Our approach is based on two key observations. First, consecutive video frames are often similar to each other. This property is in fact used by most video encoding formats (*e.g.*, H.265 [194]). Computing resources are thus wasted on processing large amounts of data that are similar or even identical to previously processed data. Second, many vision tasks are tolerant to small amounts of noise. Our key idea is thus to make deep networks *incremental* and *approximate*: First, we modify deep networks to get them to perform inference using as input only changes in pixel values between consecutive frames rather than entire frames. Second, our approach processes only changes that are sufficiently *significant* to reduce computation load for insignificant "noise" that does not notably affect model output. Incremental processing has been used in many contexts in database systems, including incremental view maintenance [96, 98, 168, 173] and semi-naive datalog evaluation [50, 96, 208]. We apply this idea to video stream processing with deep networks.

The above two observations have also been utilized by other work. NoScope [124], for example, filters input video frames using a *difference detector*, which computes the distance between two consecutive frames and only passes a frame when the distance is above a fixed threshold. NoScope focuses only on binary classification problems, such as: "is there any car in this frame or not?" It first runs a smaller *specialized* neural network model, which is selected among a set of candidate models that have different structures and are pre-trained offline, to get the result as a *confidence* score for a frame passed by the difference detector. The reference model is called only when the confidence is within a pre-defined range (*i.e.*, neither too high nor too low). Our approach, however, is different. First, our goal is to design an adaptive system that can process dynamically changing input online without much offline training and hyper-parameter tuning overhead; second, we aim at generating the same output as the reference model, no matter how complex its result is, instead of only focusing on binary classification problems; third, our delta-based optimization technique applies to not only the input frames but also operators inside of a model, therefore it can be used to reduce computation load for any convolutional neural network models, including both specialized models and reference models. A system such as NoScope may use our approach to further improve its performance.

In this chapter, we present Deluceva, a system that accelerates neural network evaluation for video analytics using delta-based inference. Figure 4.1 shows the overall architecture of Deluceva.

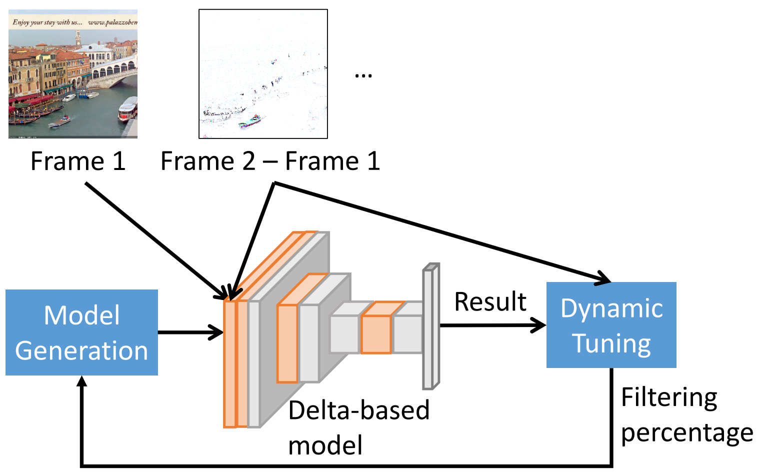


Figure 4.1: The architecture of Deluceva. A video stream is decomposed into a list of frames and evaluated on a variant of a pre-trained model. Starting with an initial filtering percentage, a delta-based model variant is generated, in which operators can operator on either deltas (orange) or actual values (white). A frame is evaluated on the delta-based model, and its result is used to tune the filtering percentage, which affects model generation for the next frame. Except the first frame, only the deltas between consecutive frames are processed. The process repeats iteratively.

It takes a video stream as input and evaluates it on a reference model pre-trained for tasks such as object detection on images. To process deltas, Deluceva automatically generates a delta-based model variant that has the same high-level structure and weights as the reference model but can operate on deltas. The video stream is decomposed into a series of frames. The first frame is fully evaluated to initialize the system, but for later frames, only deltas between two consecutive frames are sent to the pipeline. Inside the model, an operator may take deltas (orange) as input or actual values (white) and only sends significant deltas to its downstream operators that may also take takes as input. Such an operator is called a *delta-based* operator in contrast to its variant that operates on actual values. The type of an operator is determined by our tuning algorithm according to two factors: the *filtering percentage*, a value computed at the end of the previous evaluation that determines which deltas to consider significant, and the operator's performance characteristics. Finally, the model output is evaluated, and its error serves as feedback to tune the filtering percentage for future frames. The process repeats iteratively.

The Deluceva design raises several technical challenge. First, we need to develop a method that automatically transforms a deep network into one that processes delta-frames instead of full frames, and does so efficiently. For this, we need to enable deep networks to process sparse, delta-based tensors. While there exist efficient libraries for sparse linear algebra [73, 74, 97], we observe that we can achieve better performance by developing more specialize operators that leverage the specific characteristics of deep learning, such as kernel depths. Our first contribution is thus to design and implement several sparse tensor operators specialized for deep networks, including sparse convolution and a filtering operator that builds a histogram and filters deltas according to the filtering percentage (Section 4.2).

The second important challenge lies in automatically deciding when changes in individual pixel values characterize as a significant delta. We develop and evaluate two approaches to solving this problem (Section 4.3). First, we model the problem as a proportional-integral controller and dynamically adjust the filtering percentage based on the observed output errors. We do so in a way that keeps overhead low while ensuring high output quality. Second, we model the problem as a learning problem and develop a linear model that predicts what changes are significant based on the content of a delta frame.

Third, it is well-known that sparse operators, outperform their dense counterparts only if their input is sufficiently sparse. Therefore, we show how to generate a neural network model variant that consists of mixed-type (dense or sparse) operators and we derive a simple criterion for choosing what type of operator to use (Section 4.4).

Finally, we implement our approach in TensorFlow [39] and evaluate it on three representative object detection models and six videos. Our approach dynamically adjusts the filtering percentage and generates model variants. The evaluation shows that our approach outperforms the original model inference by up to 79% in terms of total compute resources while keeping object detection errors, measured using F_1 scores, below 0.1 (Section 4.5). In summary, we make the following contributions:

- We show how to accelerate neural network model inference for video analytics through incremental processing. We design and implement several efficient sparse tensor operators including convolution and a new filter operator (Section 4.2).

- We develop algorithms to dynamically adjust the amount of computation for video analysis models using either a PI controller or a machine learning model (Section 4.3).
- We show how to generate a neural network model variant that consists of mixed-type (dense or sparse) operators and present an approach to automatically selecting the operator variant to use (Section 4.4).
- We evaluate our approach on three representative object detection models implemented in TensorFlow [39] and six videos and demonstrate significant performance gains compared to the current approach, which processes frames independently (Section 4.5).

Although we focus on object detection models in this work, our approach is applicable to other applications based on convolutional neural networks.

4.1 Background

We review neural network models, particularly object detection models, for image and video analysis.

Computer vision and neural networks: The history of computer vision dates back to the 1960s. Classic techniques use image-specific or task-specific features, such as SIFT [147] and HoG [228]. Many later methods [82, 88, 89, 203] are built on top of these classic techniques. More recently, models based on neural networks have become the state-of-the-art for computer vision tasks. A neural network consists of many connected *operators* that simulate human neurons. Each operator takes and processes inputs (as neurostimulation) and generates outputs, both are usually high-dimensional tensors. For vision tasks, *convolutional* neural networks are commonly used because convolution effectively combines local information. Besides convolution, common operators include *rectifier linear unit (ReLU)*, which is used as the neuron activation function, and *pooling*, which reduces input and output sizes by performing local aggregations such as *max*.

Figure 4.2 illustrates the approach with a highly simplified convolutional neural network consisting of a few representative operators. The shape of an operator's output tensor is determined

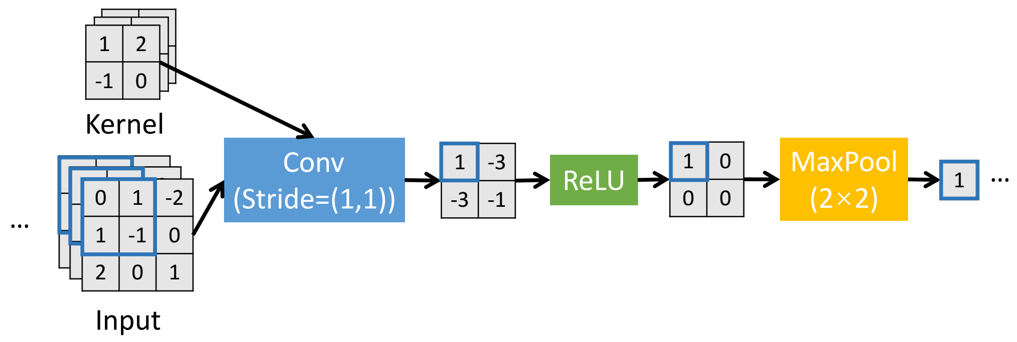


Figure 4.2: A slice of an example convolutional neural network model with a few representative operators. `Conv` **applies the $2 \times 2 \times 3$ kernel to each $2 \times 2 \times 3$ sub-input-tensor.** `ReLU` **outputs $max(0, v)$ for each input scalar v.** `MaxPool` **outputs the maximum of each 2×2 sub-input-tensor.**

by the input tensor and the operator. At the first layer, the input tensor usually contains the image pixel values (or images if batched) in different color channels (*e.g.*, RGB). The convolution operator applies the $2 \times 2 \times 3$ kernel to each $2 \times 2 \times 3$ sub-input-tensor and outputs their inner product. Moving the inner product within the input tensor with strides of $(1, 1)$, the convolution generates an output tensor with shape 2×2. `ReLU`, which serves as the neuron activation function in this example, outputs $max(0, v)$ for each input scalar v. The pooling operator `MaxPool`, with a 2×2 kernel size, outputs the maximum of each 2×2 sub-input-tensor. Although not directly shown in the figure, in the end, neurons at the highest layer with the most significant activations are used to generate model outputs.

Deep neural networks: A neural network is *deep* if it has many layers. Compared to earlier methods, deep neural networks require more computation since a model usually contains many layers of expensive operations such as convolutions. This amount of computation used to be unaffordable, but thanks to advances in modern computing devices, it is now possible to train and evaluate these models within a reasonable amount of time. Deep neural networks are powerful because they can enumerate all potential features corresponding to every small local region and derive high-level features from combinations of those low-level features. Different from previous methods in which features are manually designed for certain tasks, in deep neural networks, important features for a certain task are automatically chosen from a large space of possible features through gradient-descent-based training on large datasets. Deep neural networks have been shown to perform extremely well on image-based object classification and object detection tasks [88,132,144,189,198].

Object Detection Models: The detection of objects present in images (or video frames) consists of two tasks: object localization and object classification. Many object detection models have recently been proposed [68,88,89,144,171,172,181]. Although they have different architectures and levels of accuracy, in general, they share the same objectives of: extracting features from raw images, proposing regions that may contain objects of interest, and performing object classification within those regions. As the first step, an established convolution neural network architecture [106, 112,119,132,189,198] is usually used as the feature extractor. For the latter two objectives, some models treat localization and classification as two steps [88,89,172,181], while others [68,144,171] combine them into one for fast inference.

Model Inference: Because of the success of neural network models on images, most video analysis approaches apply image models directly to each video frame [105,124,125,171,192]. However, when evaluating image models, people care more about accuracy than speed. In competitions such as ImageNet [15] and COCO [142], accuracy is the only metric of interest. Recently, there have been models developed specifically for fast inference [68,112,144,171], and people have noticed the importance of inference speed [114]. However, much room remains for optimization if we aim at analyzing large numbers of high-resolution video streams using complex models.

4.2 Delta-Based Models

The key idea behind Deluceva is to accelerate model inference over video streams by eliminating unnecessary computations. We model a video as a stream of frames and leverage the observation that neighboring frames in videos are similar to each other. Instead of processing full frames, we take the differences in pixel values between adjacent frames and only send those delta tensors to a new type of delta-based model for inference. Internally, delta tensors are filtered and processed by delta operators. Figure 4.3 illustrates the approach. Given a model, the top row shows inference performed on one frame, Frame 2, using the original model. The second and third rows show the delta-based approach. The first frame, Frame 1, is processed using our modified delta-based

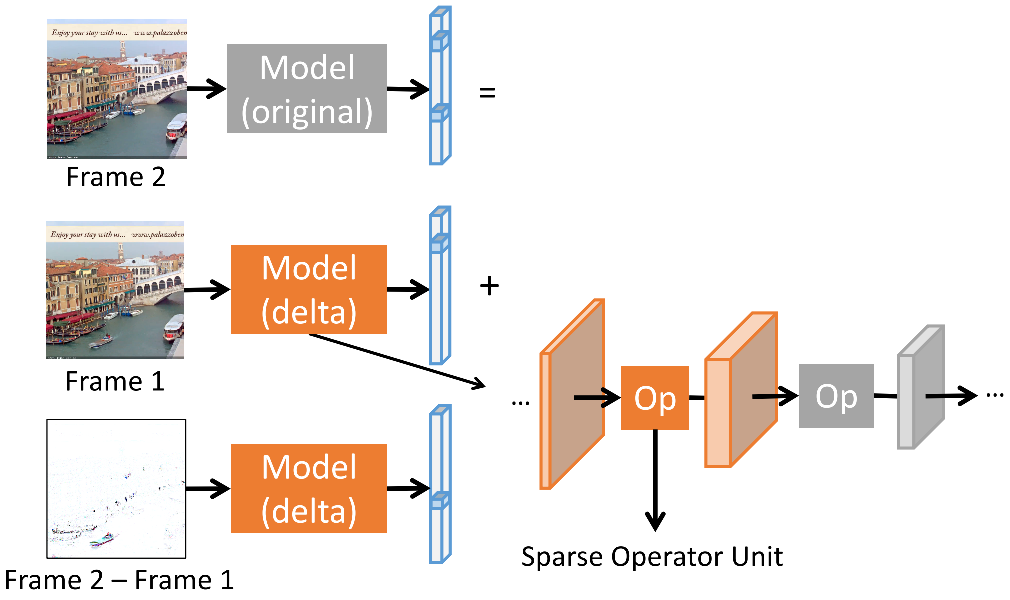


Figure 4.3: Example incremental network. First row: model inference for Frame 2 using the original model. Second and third rows: adding the outputs of the delta-based model for Frame 1 and (Frame 2 - Frame 1). On right: Inside of the delta-base model, operators are either dense (gray) or sparse (orange), and they work with either delta (orange) or non-delta (gray) tensors. The internal structure of a sparse operator unit is presented in Section 4.4.

model, in which state is kept internally as necessary. Subsequent frames are then processed by sending deltas between two consecutive frames to the model. The output of the delta-based model is combined with the output produced so far to form the output for the latest frame.

The delta-based model may consist of both delta and non-delta operators. Formally, we define an operator to be a *delta* operator if it takes delta tensors as input and produces delta tensors as output. We denote a delta operator as op_d and its non-delta variant as op. When evaluating a frame F_i, each operator takes one or more input tensors and generates one or more output tensors. If I_i and I_{i+1} are the inputs of the original variant op when evaluating two consecutive frames F_i and F_{i+1}, then we define op_d as:

$$op_d(I_{i+1} - I_i) = op(I_{i+1}) - op(I_i), \tag{4.1}$$

where $I_{i+1} - I_i$ is the delta tensor.

A naive approach to implementing the above type of delta operators would be to encode their inputs and outputs using a dense representation, where a delta tensor is an n-dimensional array filled with either zeros or delta values. This encoding enables the direct use of common tensor operators that process dense inputs and produce dense outputs. However, processing deltas encoded as dense tensors does not save any computation because zero-valued deltas are still being processed. Instead, we need efficient operator implementations that operate on delta inputs that contain fewer values than the dense tensors. Specifically, we require the above delta operator to take a sparse representation of deltas as input, and its time complexity to be linear in its input size, to benefit from our delta-based processing approach.

While the conceptual idea is simple, the key technical challenge lies in efficient implementation and integration with existing deep learning libraries. The literature on efficient sparse-dense linear algebra is extensive. In particular, sparse-dense matrix multiplication is a key operation, and many libraries [73, 74, 97] have highly optimized implementations. The optimizations can be grouped into several categories including: sparse matrix compression using structures such as blocks and triangles [86, 117, 136, 211], register-level and cache-level blocking [117, 166, 200, 206, 222], and SIMD vectorization on modern architectures that support vector instructions [211, 222]. However, the integration of these optimizations with popular deep learning libraries is currently limited. In this work, we use TensorFlow [39] as our reference framework. While TensorFlow does provide sparse versions of some operations such as matrix multiplication, it does not have efficient sparse versions of other key operations, such as convolution for example.

To address this limitation, we implement new sparse operators. Our goal is to use simple optimizations, so as to facilitate the implementation of large numbers of sparse operators, yet ensure their efficiency to yield improvement over their dense counterparts. In this section, we present our design for the sparse convolution operator in detail as it is one of the most widely used and most expensive operators. As we show later in Section 4.5.1, convolutions are the most frequent operators and account for the majority of inference time in all three object detection models that we evaluate. Other common operators can be similarly implemented.

Convolution is usually implemented using multidimensional matrix multiplication since tensors are multidimensional arrays. Particularly, sparse convolution has sparse matrix-dense matrix multiplication at its core. However, when the stride is less than the kernel size, each input scalar is involved in multiple kernel applications, so we need an additional costly transformation (called *image-to-column* (`im2col`) in some libraries [14]) to duplicate and rearrange input scalars before using matrix multiplication. Due to its overhead, it is not always efficient to implement sparse convolution using sparse matrix multiplication. In this section, we introduce two designs of the sparse convolution operator, one is for running on CPUs and the other is for GPUs.

For the CPU implementation, we were able to obtain better performance by implementing a sparse convolution operation directly rather than relying on TensorFlow's sparse matrix operations. Our implementation leverages the key observation that, in deep networks, the kernel tensor of a convolution has a depth dimension for stacking multiple kernels, so we can use this dimension to trigger SIMD vectorization and achieve high performance, competitive with dense implementations. The right half of Figure 4.4 shows the internal structure of our sparse convolution operator (we discuss the left-hand side of the figure in the next section). It takes a sparse tensor, which consists of a list of indices and values, as input. Each pair of (index, value) represents a scalar. An input scalar contributes to multiple positions of the output tensor, which are determined by the kernel shape. [1] In this example, the kernel size is 2×2, and two kernels are stacked to form the depth dimension of the kernel tensor. The output tensor has the same depth as the kernel tensor. Each input scalar s maps to one or few positions of the kernel and output tensor, each mapping leads to two vectors, the kernel vector $\boldsymbol{k}$ and the output vector $\boldsymbol{o}$, as shown in Figure 4.4. The computation ($\boldsymbol{o} = s \times \boldsymbol{k} + \boldsymbol{o}$) is done using two vector operations: scalar-vector multiplication and vector-vector addition. The complexity of processing one scalar is then linear in the kernel tensor depth. In order to vectorize both operations, we use multidimensional arrays to store both the kernel and the output tensor and align both arrays to ensure that the compiler generates vector instructions for them. If there are nk kernels and the degree of vectorization is d, then the number of instructions per scalar

[1]Other factors are strides and padding. We assume strides to be one and no padding in this case and omit their discussion for simplicity.

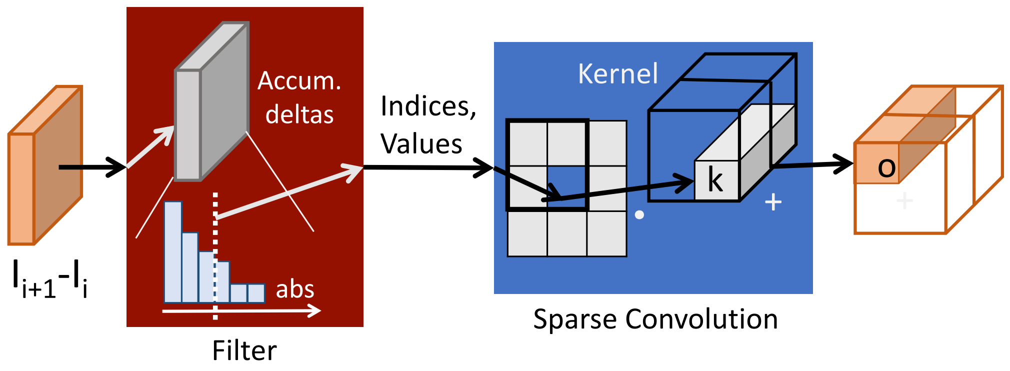


Figure 4.4: A `filter` operator followed by a sparse convolution operator. The `filter` takes a dense form of delta values, builds a histogram on inputs together with accumulated values, keeps smaller values in its state according to a filtering percentage, and sends larger values out as a list of indices and values. The sparse convolution operator takes the list as input and updates its output tensor for each input scalar using vector instructions. The time complexity of processing one scalar is linear in the kernel depth.

is $\lceil 2 \times nk/d \rceil$. [2] As a result, the output tensor is in dense format. In Section 4.5.1, we evaluate our sparse convolution implementation and characterize when it outperforms its dense counterpart.

We find, however, that leveraging `im2col` is important for an efficient sparse convolution implementation for GPUs. Because of the high degree of computing parallellism, it is important to avoid simultaneous atomic writes to the same memory location by multiple GPU threads. Therefore, for each output scalar of a sparse convolution, we assign a GPU thread to be fully responsible for computing its value to avoid having atomic operations scattered across multiple threads. As a prerequisite step, `im2col` gathers corresponding input values for each output scalar.

Figure 4.5 shows our design for sparse convolution on GPUs. The implementation has two steps. The first step is a parallel `im2col` on the sparse input tensor, which stores the output into an intermediate tensor. The intermediate tensor is divided to multiple regions, each holds input values for one output scalar. Each input scalar may have impact on multiple output scalars and thus be copied to multiple regions. In the second step, each thread responsible for computing one output scalar collects data from the corresponding region and performs an inner product between the input

[2] The last instruction may not be a vector instruction if nk is not divisible by d.

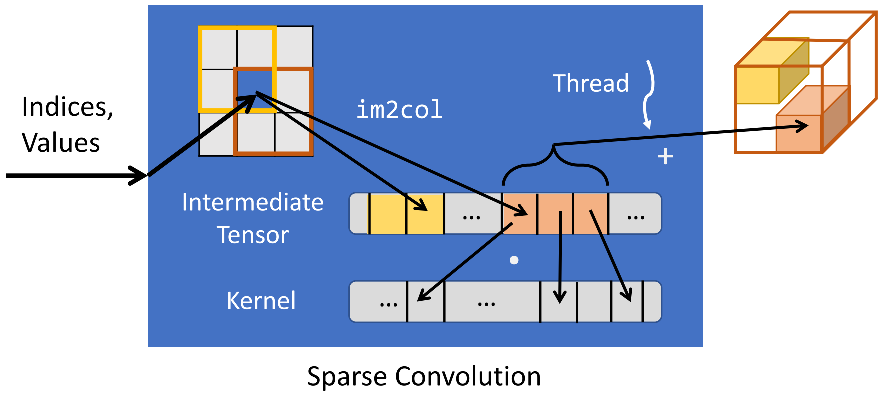


Figure 4.5: The internel structure of the sparse convolution operator for GPUs. The first step is parallel `im2col` on the sparse input tensor, which outputs to an intermediate tensor divided to multiple regions, each holds input values for one output scalar. In the second step, each GPU thread computes one output scalar by performing an inner product on the input tensor and the kernel tensor.

tensor and the kernel tensor. We evaluate our implementation in Section 4.5.1.

Other operators can similarly be implemented. Some operators also need to maintain state. For example, ReLU outputs $max(0, v)$ for each input scalar v, where v can be the actual value of a neuron activation, instead of the delta between two activations, in the original model. When ReLU is used in our delta-based model with its input being a delta tensor, it needs to maintain an internal state consisting of the accumulated deltas to operate on the actual values instead of deltas. Because of the overhead of maintaining and updating state, some operators may not have simple efficient delta-based variants and may thus be most efficient in their original non-delta-based, dense format. Our approach supports such cases as we describe further in Section 4.4.

4.3 Dynamic Tuning

To improve inference time, our approach goes beyond the idea of processing delta frames: It processes delta pixel values only once those values have *significantly* changed. Otherwise, it accumulates changes to pixel values until they pass the significance threshold. This approach further reduces the amount of processing done for individual frames and we describe it in this section.

The significance threshold is expressed as a *percentage value*: i.e., the percentage of delta values that should be accumulated in internal state. We choose to use percentage values because the absolute values depend on the content of the video itself. That percentage value is determined by our dynamic tuning algorithm that we describe later in this section.

First, we assume the significance threshold to be given and describe the filtering step. To filter away insignificant changes, we insert special `filter` operators in front of expensive operators, such as convolution, in the deep network. Filters keep deltas values below the significance threshold in their internal state, and only output large delta values. The left half of Figure 4.4 illustrates the internal structure of the `filter` operator. The operator takes delta values as input and updates its internal state of accumulated deltas. It then builds a histogram on the absolute values of the accumulated deltas. It uses that histogram to translate the percentage of delta values to retain into an absolute delta-value threshold. It then uses that threshold to keep smaller delta values in its state and send the remaining, larger values, to the downstream operator. Each `filter` can have its own filtering percentage, but in this work, we assume that there is one global filtering percentage that is used by all `filter` operators for simplicity.

The filtering percentage affects both the model output quality and the inference time. As a motivating example, Figure 4.6 shows how different filtering percentages lead to different results. Both frames are from the same video, but one of the boats in Frame 1 leaves the picture in Frame 2. The model successfully identifies two boats in Frame 1, which is the expected result. For Frame 2, we expect only one boat to be recognized. Now we evaluate Frame 2 based on states generated by Frame 1 using deltas. The first row uses a high filtering percentage, so most deltas are kept internally and only a small portion of significant deltas from each sparse operator have affected the final output. Since the previous output has two boats, in this case the model still thinks that the left boat exists and reports two boats. This indicates that the filtering percentage is too high. The second row uses a low filtering percentage, so most deltas have impacted the output. The model successfully recognizes that there is only one boat. However, its runtime, compared to the first row's runtime, is higher because more data needs to be processed. This example demonstrates how we want to balance between accuracy and speed. We would like to filter out as many insignificant

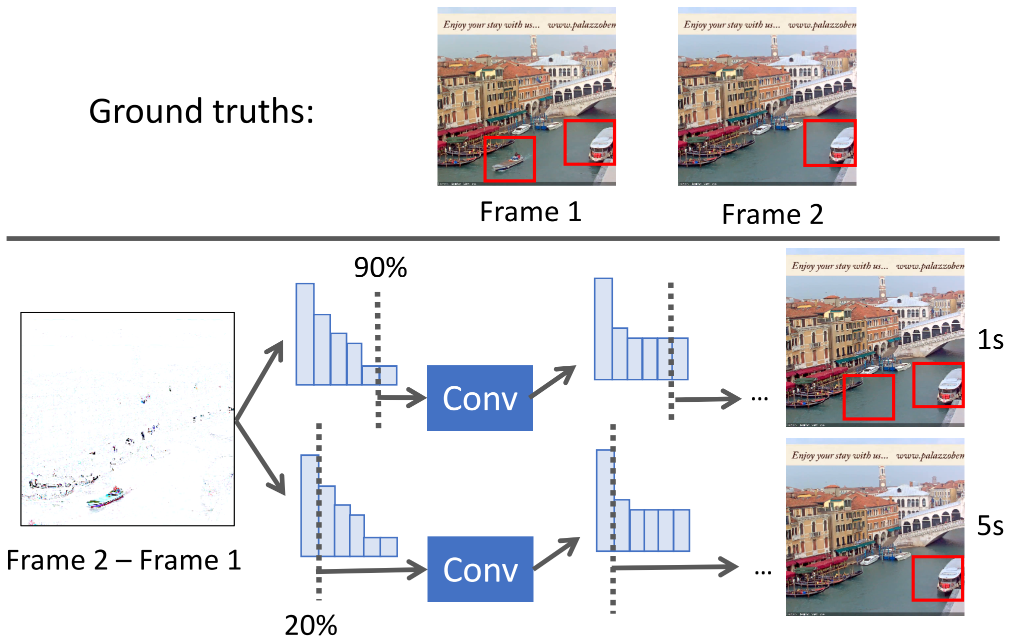


Figure 4.6: Effect of significance threshold on inference quality in a delta-model. Ground truth for Frame 1: two boats, Frame 2: one boat. The first row uses a higher filtering percentage and has a lower runtime but an inaccurate output. The second row uses a lower filtering percentage and has a higher runtime but an accurate output.

deltas as possible, while meeting a minimum desired quality threshold. In the rest of this section, we present two approaches for finding and updating that threshold.

Model Output Quality: We can measure the quality of a model by comparing its output with the ground truth. One way of getting the ground truth is to get humans to annotate videos, but this approach does not scale. We seek an automated process instead and use the output of the original dense model as the ground truth. Of course, acquiring that ground truth during video processing will be expensive and our approach takes that cost into account.

The output of an object detection model consists of a set of recognized objects, each marked with a bounding box and a class label. Given two model outputs, we say that two objects from each are the same if they have the same class label and the difference between their bounding boxes is within a threshold. Using this definition, and the output of the original dense model, we can compute a precision, p, and a recall, r, for our delta-model. We can then define the error metric using the F_1

score:

$$F_1 = 2 \cdot p \cdot r/(p + r). \tag{4.2}$$

A higher F_1 score means that our system produces results more similar to the ground truth. But, as a trade-off, it usually means that the filtering percentage is lower so that more deltas are processed. On the other hand, a higher percentage makes our system faster, but we risk producing less accurate results. Because of the impact of the filtering percentage on F_1, we can treat the F_1 score as a function of percentage $F_1(p)$. $F_1(0) = 1.0$ because our model outputs exactly the same results as the original model; And $F_1(p) <= F_1(0)$ for any $p \in [0, 100]$ since any filtering will likely bring the F_1 score down.

The next step is to find a good filtering percentage for the next frame given the error metric. We set a constant, t_{F_1}, as our target for F_1 score. Our goal is to find the highest filtering percentage that produces an F_1 score no lower than the target, and we call it the *target percentage*. Interestingly, we observe, that dense models do not always output correct results. In some cases, an object being recognized by a dense model on the previous frame is ignored on the subsequent frame even if the two frames are similar. In some cases, our sparse model is still able to recognize the object because it processes only large-enough deltas. We posit that this is because the dense model is sensitive to noise while our sparse model works as a noise filter. This observation means that aiming for an F_1 score of 1.0 is not necessarily desirable. In our experiments, we find that $F_1 = 0.8$ yields good results.

The Ideal Approach: As baseline, we first describe an approach that finds the optimal target percentage for each frame by processing the frame using both the original dense model and the new delta-model. This approach enables us to determine an upper bound on the performance gains that we could observe for a given model and video combination.

The key idea of the approach is to first process a frame with the original dense model to get the ground truth. The approach then performs a binary search on the percentage target. For each threshold value that it tests, it records the resulting F_1 measure. The optimal target percentage is the lowest value that exceeds the desired F_1 metric.

F_1 and the filtering percentage p have an implicit relationship: higher ps usually bring lower F_1

scores while lower ps usually lead to higher F_1 scores. The function $F_1(p)$ is not guaranteed to be monotonic, because a model may output more objects even when fewer deltas are being processed. However, based on our observation, monotonicity generally holds. So in this work, we empirically assume its monotonicity and use binary search to find the target percentage. An alternate solution would be to try all p values exhaustively.

This Ideal Approach is, of course, impractical because it processes each frame with both the original dense and our new delta-model. Additionally, it performs binary search on the percentage target, which requires several evaluations of our delta-model. Assuming the search stops when the length of the current search interval is below 1%, then the binary search needs 7 model evaluations.[3]

We use the Ideal Approach as a reference point and develop two others approaches to find the target percentage with a small overhead. The first one models the problem as a control problem and uses PI controller (Section 4.3.1) to adjust the ratio dynamically. The second one trains a machine learning model offline and uses the model to predict the target percentage for future frames (Section 4.3.2). We present both approaches in the rest of this section.

4.3.1 PI Controller

In this approach, we model the problem of finding the target percentage as a control problem. In a control system, a controller monitors a controlled process variable and compares it with a set point. The error between the process variable and the set point is applied as feedback to adjust the output of the controller, in order to make the controlled variable close to its set point in a dynamic environment.

Among all common controllers, proportional–integral controller (PI controller) is a representative controller mechanism widely used in control systems. [4] At each time t, it calculates an error value $e(t)$ and uses a function with two terms: a proportional term and an integral term, to determine the

[3]In our implementation, we do binary search by starting with the percentage of the previous frame, instead of 50%. The overhead is reduced but still larger than one model evaluation.

[4]A more general form is a proportional-integral–derivative (PID) controller, but the derivative term is known to be sensitive to noise [25]. We omit the derivative term in this work.

future controller output. Formally, the controller output at time t, $CO(t)$, is defined as:

$$CO(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(t), \tag{4.3}$$

where K_p and K_i are two parameters that need to be tuned. The controller output thus depends on the instantaneous error observed in the last time-step and on the errors accumulated over time.

In our case, our goal is to keep the F_1 score close to the target t_{F_1} (the set point). Our time is discrete since we adjust the target percentage between frames. Specifically, we set the filtering percentage for frame $t+1$, $P(t+1)$, to be:

$$P(t+1) = P_{bias} + K_p \cdot e(t) + K_i \cdot \sum_{k=1}^{t} e(k), \tag{4.4}$$

where $e(t) = F_1(P(t)) - t_{F_1}$ and P_{bias} is a constant that enables our system to start from a reasonable target percentage as it processes the first frames.

One known issue of PI controllers is *reset windup*. When the system decides on a controller output that exceeds the controller's maximum (or minimum) output, it may take a long time (or never) for the system to reach the set point since the controller is not capable of producing the desired value. During this period, large errors may accumulate and the integral term may continue to grow with no limit. In our case, the range of the controller output P, is $[0, 100]$. Since $F_1(0) = 1.0$ and $t_{F_1} \in [0, 1]$, reset windup does not happen on the lower end. But, it may happen on the higher end since it is possible to have $F_1(100) > t_{F_1}$. For example, when the current frame is exactly the same as the previous one and the previous output matches the ground truth, we have $F_1(100) = 1.0$ because the result is still the same as the ground truth even if we do not process any new data. We solve this issue using a standard solution: We do not integrate the error $e(t)$ if the controller output has exceeded its maximum, *i.e.*, $P(t) > 100$. Certainly, when a filtering percentage is actually used to generate a deep network, we always truncate it to be within $[0, 100]$.

Using a PI controller brings overhead. The overhead comes from running the original model to get the ground truth for a frame. Since we only adjust the percentage when we have a new error, we want to calibrate frequently enough such that the filtering percentage is changed on time but the overhead is small. In order to make this approach faster than directly running the original model, we only do one calibration every few frames and amortize the cost over them. As we show later in Section 4.5.2, the calibration frequency is an important factor of the overall performance: Higher frequencies have larger cost but may produce more accurate results, while lower frequencies lead to higher errors in some cases.

4.3.2 Machine Learning Model

Another approach to setting the target percentage is to predict it by using a machine learning model. Intuitively, when two frames are mostly the same, it is safe to set the percentage to be high since we expect most neuron activations to be similar. On the other hand, when the delta between two frames is large, we may need to lower the percentage to make sure changes are reflected in the model output. The key idea of our approach is thus to extract features from the delta-frame and use those features to predict a target percentage for the new frame. Additional features may also be useful, in particular, we include the previous predicted target percentage for the previous frame.

We thus develop a machine learning model for predicting the target percentage for each frame. We include two types of features: the predicted percentage for the previous frame and several statistics computed over the delta pixels between the previous and the current frame. Since many moving objects do not occupy a whole frame, to better capture local changes, we divide the image into several overlapping tiles and collect statistics from each. Specifically, we compute the absolute values of pixel differences in a tile and use the sum and the max as the features of the tile.

The model needs to be trained on frames with known target percentages. However, as presented above, finding the optimal target percentage is expensive, so we perform the training offline: We take a short video snippet and compute, for each frame, the target percentage using the Ideal Approach together with the features for the model. We train a linear model. Formally, If an image is divided into $R \times C$ tiles, the length of the feature vector $\boldsymbol{w}$ is $2 \cdot R \cdot C + 1$. The model outputs:

$$P(t+1) = \boldsymbol{x}_{t+1}^T \cdot \boldsymbol{w} + \boldsymbol{b}, \tag{4.5}$$

where $\boldsymbol{b}$ is the bias and $\boldsymbol{x}_{t+1}$ is the feature values of frame $t+1$. We get the values of $\boldsymbol{w}$ and $\boldsymbol{b}$ by minimizing the root-mean-squared error on the training set. Same as before, $P(t+1)$ is later truncated to be within $[0, 100]$. There exist other models, but as we show later in Section 4.5.4, a linear model is sufficient for predicting target percentages.

The major drawback of this approach is the large overhead of getting the training data as it requires computing the target percentages using the Ideal Approach, which performs a binary search on possible percentage values. If we amortize the cost over the frames being evaluated, then this approach is only faster than running the original model if we train the model on a short video clip and use it to evaluate a much longer video.

We denote with f the training-to-test factor, which is the number of training frames over test frames, and experiment with different f values. As we show later in Section 4.5.4, the size of the training set is important: The model produces incorrect predictions when the training set does not contain enough frames; However, the amortized cost is high if the training set is too large. As the video stream changes (e.g., day turns into night), retraining is also necessary to achieve good results. We present the detailed evaluation results in Section 4.5.4.

4.4 Mixed Network

The final component of our approach is the end-to-end delta-model: Given the original model, we construct a model variant that processes deltas efficiently. A straight forward solution would be to replace all dense operators in the original model with their delta-based, sparse variants. This approach, however, is not always possible and, when possible, is not always the most efficient solution. Instead, our approach is to mix sparse and dense operators and construct a mixed network.

The first reason to not always use a fully delta-based network is that we may not have a sparse version for a less common operator. Another more important reason is that a sparse operator can sometimes be slower than its dense alternative even when its input is filtered according to a high filtering percentage. We observe that there are two key issues that may have negative impact on the performance of our sparse operators.

The first one is the need to have the actual values instead of only deltas. We use the Rectifier function (ReLU), which takes an input tensor t and outputs $max(0, t)$, as an example. Since ReLU outputs the larger one between 0 and the actual value, only knowing the delta is not enough. We need to reconstruct the actual values by adding deltas and the previous values together, which takes a few more instructions per delta input.

The other issue is vectorization. In a dense ReLU, the `max` can be done using vector instructions. However, in a sparse ReLU, in order to get the actual values, we need to do index lookups to add deltas to previous values. This operation is not always vectorizable. Some modern architectures support instruction sets such as Intel's Advanced Vector Extensions 2 (AVX2) and 512 (AVX-512), which have vector instructions for scattering and/or gathering between memory and registers using indices, but others may not have support for it. Even in these instruction sets, scatter and gather have much larger latencies than other basic operations. In summary, we may lose the significant speedup if index lookup is required for an implementation.

Fortunately, convolution is an operator that may benefit from sparse inputs. Since convolution is a linear function, it produces the delta of the outputs of two convolutions by performing convolution on the delta of its input directly without the need to maintain its previous state. It is also possible to vectorize its computation on the depth of the kernel even when we cannot vectorize index lookups, as we described in Section 4.2. But other operators, such as pooling and normalization, have similar issues as ReLU since they all need current values and are not always vectorizable if the input is sparse.

To address the above challenge, our approach is to mix sparse and dense operators to construct the delta-based network. To connect the two types of operators, we need to convert between current values and deltas. We implement two special operators: `d2c` for converting deltas to current values and `c2d` for converting current values to deltas. These two operators, together with `filter`, are inserted by our algorithm when needed to connect sparse and dense operators and to filter out insignificant deltas. [5] We call a sparse operator, together with the other three operators, a *sparse operator unit*, which is used to compare against its dense version. Figure 4.7 shows a sparse operator unit together with its dense variant.

Given the current filtering percentage, for each operator in the model, we decide if the dense

[5] `c2d` and `d2c` can be removed when two consecutive operators are both sparse or both dense.

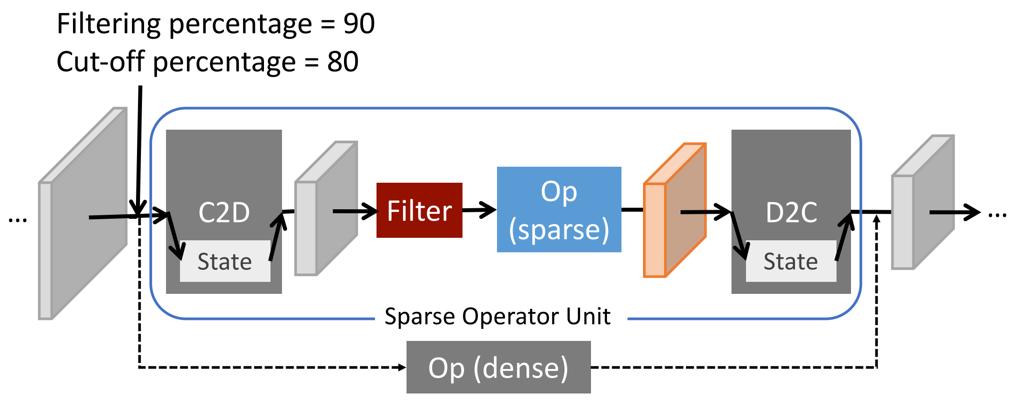


Figure 4.7: An operator's dense variant and its sparse operator unit. The sparse operator unit is chosen as the faster implementation in this case since its cut-off percentage (80) is lower than the filtering percentage (90). It is further connected with other operators in the network.

alternative or the sparse operator unit should be used. The decision is made according to the operator's runtime characteristics: We profile both variants using different filtering percentages and find the percentage that makes their runtimes close to each other. We call it the *cut-off* percentage. The mixed network is then generated accordingly: We use the sparse operator unit for an operator if the current filtering percentage is over its cut-off percentage, otherwise we use its dense implementation. For example, in Figure 4.7, the sparse variant is selected because the filtering percentage is 90 and the cut-off percentage of this operator is 80, which indicates that the sparse version is expected to be faster that the dense variant. We present the results in Section 4.5.1.

4.5 Evaluation

We evaluate Deluceva using three object detection models implemented in TensorFlow [39]. All experiments are done on Amazon EC2 `r3.2xlarge` instances. We evaluate all models using one thread on one CPU to most accurately compare performance differences between approaches without variance due to the TensorFlow scheduler. We exclude the time of TensorFlow's graph optimization from each evaluation.

The three models are summarized in Table 4.1. Each model is built on a network architecture designed for object detection and uses a neural network model as the feature extractor. More

Name	Architecture	Feature Extractor	Size	# of FLOPs	Time (Original, Fully Delta)
`ssd-vgg`	SSD	VGG-16	Small	123.4B	3.6s, 7.6s
`frcnn-res`	FRCNN	ResNet-101	Medium	550.6B	16.2s, 39.8s
`frcnn-incep`	FRCNN	Inception-ResNet-V2	Large	1395.0B	41.2s, 187.4s

Table 4.1: The three object detection models used in the experiments.

specifically, as introduced in Section 4.1, for network architectures, SSD [144] combines region proposition and object classification into one step, while FRCNN [172] treats them as two steps using two sub-networks. The feature extractor networks include VGG-16 [189], ResNet-101 [106], and Inception-Resnet-V2 [197]. The model sizes are from small to large, and their total numbers of floating point operations are listed. We also provide two inference times as references: the time to run the original model (Original) and the time to run our generated delta-based model (Fully Delta), where *every* convolution operator is replaced by a sparse operator with its filtering percentage set to zero (i.e., processing all deltas). We use [28] as the reference implementation for `ssd-vgg` and TensorFlow's object detection API [114] as the reference for the two FRCNN models.

For training and test data, we capture six 10-minute videos from three YouTube live streams, each has two videos taken at different times. Their information is listed in Table 4.2. We take a snapshot every second from each video to generate frames. All frames are converted to 300×300 pixels in `png` format.

4.5.1 Delta-Based Network

Although there are many neural network operators, the most common ones are of the following types: convolution, activation function (*e.g.*, ReLU), pooling (*e.g.*, max pooling), and normalization (*e.g.*, batch normalization). Based on our observations, convolution usually dominates runtime. To verify this intuition, we study the runtime distribution of the different operator types in the three models. Figure 4.8 shows the detailed statistics of three major operators: convolution, max pooling,

Stream	Time	Abbrv.	Avg. # of Objects of Interest			Typical Objects
			ssd-vgg	frcnn-res	frcnn-incep	
Jackson Hole [17]	Evening	je	3.85	8.25	4.40	cars, people, traffic lights
	Night	jn	0.15	5.12	0.49	cars
Venice [33]	Day	vd	5.07	9.60	5.78	boats, people
	Night	vn	0.53	1.00	0.41	boats, people
Kusatsu [20]	Day	kd	4.05	11.31	7.42	cars, people, buses
	Night	kn	0.49	3.19	2.04	cars, people

Table 4.2: The six videos used in the experiments. Each video is 10 minutes long and has a resolution of 300×300.

ReLU, and also other operators. Each bar shows the total runtime of one operator type in one model, and the number of operator instances in the model is shown on top of each bar. As the figure shows, convolution takes most of the runtime and is thus important to optimize, while other operators do not have large performance impact. Because of the overhead and platform-dependent performance as we introduced in Section 4.4, we use the original implementation for all operators and only consider replacing convolution with sparse operator units.

Next, we evaluate the performance of our sparse convolution operator. We compare its runtime against the runtime of the original implementation, which takes a dense tensor as input. The sparse convolution's input is a list of delta indices and values filtered by a `filter` operator, we vary the filtering percentage to evaluate the impact of input size on runtime. For image processing, both the input tensor and the kernel are usually four-dimensional tensors with shape [# of batches, # of rows, # of columns, depth]. In the three models that we test, the numbers of rows and columns range from 1 to 600 for the input tensor and 1 to 7 for the kernel. Depths vary from 1 to 2080, and the number of batches is either 1 or 300 (as the number of proposed regions).

Figure 4.9 shows the runtimes of the sparse convolution operators over the corresponding dense ones as we vary the filtering percentage. Each curve represents one unique convolution operator. We

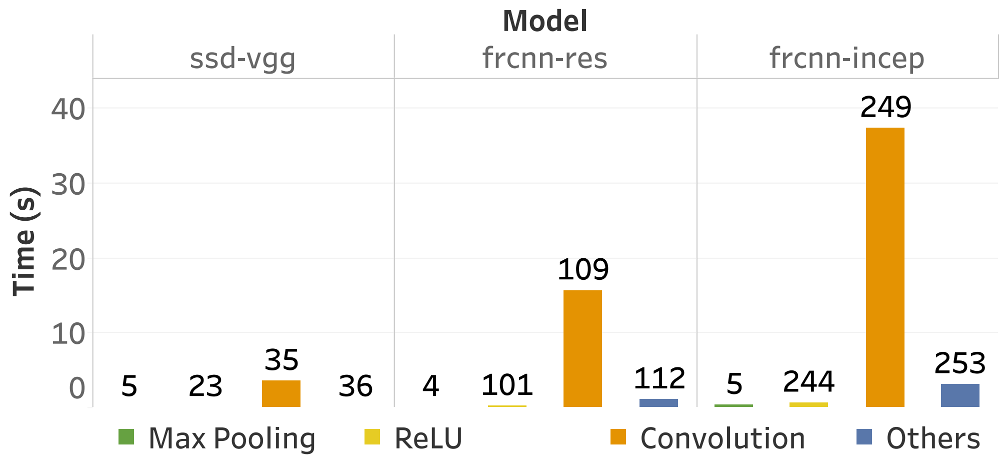


Figure 4.8: Total runtimes of common operators in the three models. The total number of each operator in each model is labeled on top of each bar. Convolution dominates runtime, other operators do not have large performance impact.

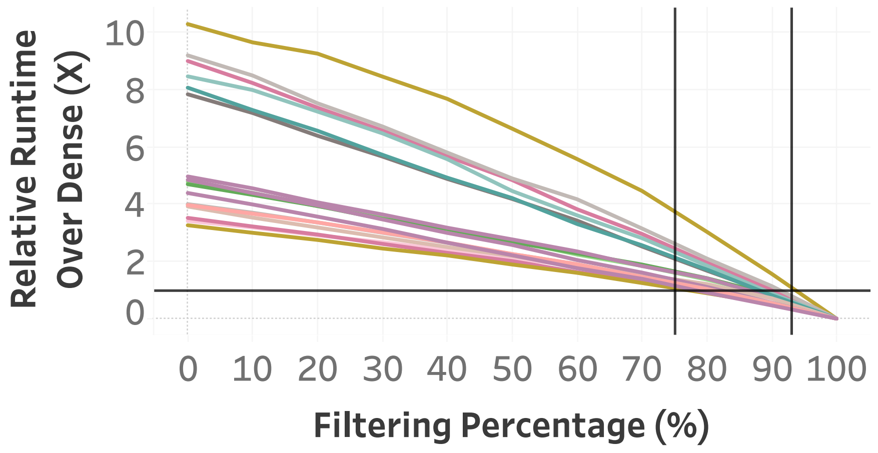


Figure 4.9: Relative runtimes of sparse convolution over dense convolution with different filtering percentages. Each line represents one convolution. The runtime is mostly linear to the input size for each convolution, but different convolution may have different slopes. The crossover range is between 75% and 93%.

treat two convolution operators as duplicates if they have exactly the same parameters, including input tensor shape, kernel shape, stride, and padding. The runtime of a specific sparse convolution is mostly linear in its input size. However, the slopes of these linear functions are different for different convolutions. This is because many factors besides input size affect runtime, including the total memory size used by the operator, cache misses, and register loads/stores. Our implementation does not target any specific optimizations besides vectorization. Other optimization are possible. For example, the GEMM-based (GEneral Matrix to Matrix Multiplication) level 3 Basic Linear Algebra Subprograms (BLAS) [123] is carefully designed to reduce data traffic in a memory hierarchy by blocking. We leave these additional optimizations for future work. Since different convolutions have different performance characteristics, our approach is to profile each shape independently instead of building a unified model for all possible shapes. To be as fast as the original dense implementation, we need to filter out 75% to 93% of the input data for a sparse convolution operator. Although the percentages are high, as we show later in this section, we are able to have significant performance gains with our prototype implementation.

The next experiment is to find the cut-off percentages for each sparse convolution unit with a unique shape. We profile each unit and compare it with the corresponding dense convolution. Figure 4.10 shows the cut-off percentages for sparse convolution units in the three models and also the runtime breakdown for the operators inside each unit (D2C, Filter, Sparse Convolution, and C2D). A cut-off percentage of 90 means that we need to filter out 90% of the deltas to make a sparse convolution unit as fast as a dense convolution. Most of the cut-off percentages are below or close to 90, but there are a few from `frcnn-res` and `frcnn-incep` that are above 90. These convolutions mostly have a large input tensor with a low-depth kernel, so that `filter` is expensive while the time saved by filtering out deltas does not dominate total execution time of the operator. The overhead of `filter` is caused by our implementation: We scan the input tensor multiple times, including one scan to build the histogram, which is not vectorized. This overhead becomes a dominant factor in these cases. Additional optimizations are also possible for the filter operator, but we leave them for future work. However, as we show later in Section 4.5.2, having these cut-off percentages is already enough to significantly accelerate model inference.

We also evaluate the GPU implementation of our sparse convolution unit. Similarly, Figure 4.11 shows the runtimes of the sparse convolution operators over dense ones with different filtering percentages, and Figure 4.12 shows the cut-off percentages of each unit, on a NVIDIA Tesla K40c GPU. Most sparse convolution units have relative runtimes over dense convolutions by a factor of

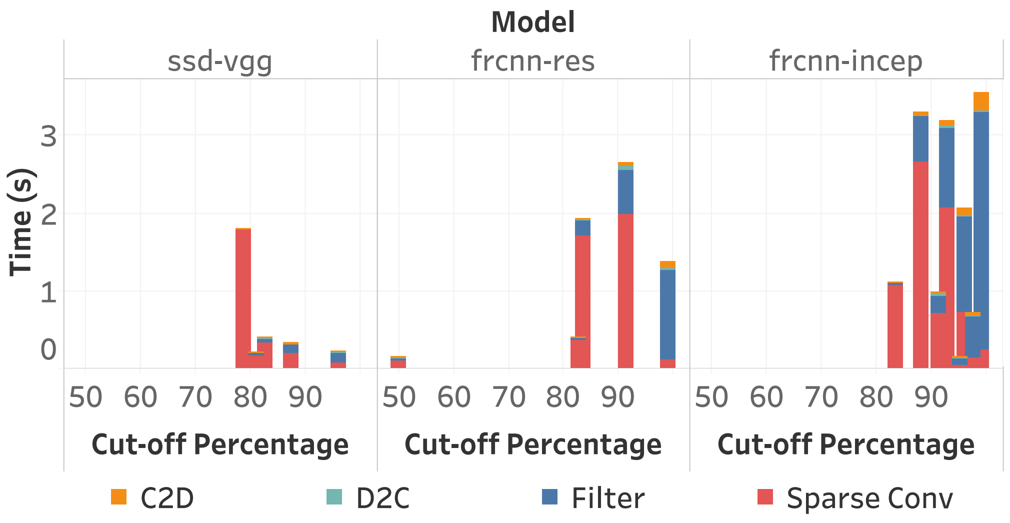


Figure 4.10: The cut-off percentages of sparse convolution units in the three models on CPU. Each bar represents one convolution unit. Each color represents the runtime of one operator in a unit.

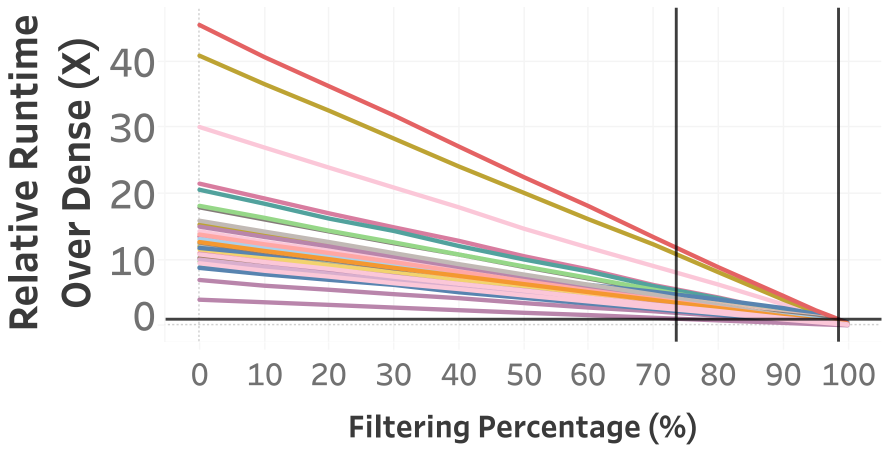


Figure 4.11: Relative runtimes of sparse convolution over dense convolution with different filtering percentages on a GPU. Each line represents one convolution. The crossover range is between 73% and 98%.

20 or below, while a few ones reside between 20 to 45. The crossover range for cut-off percentages is between 73% to 98%. Compared to the CPU version, the GPU implementation has higher relative runtimes over dense, but we still get positive savings in many cases. We believe further optimizations remain possible and leave it for future work.

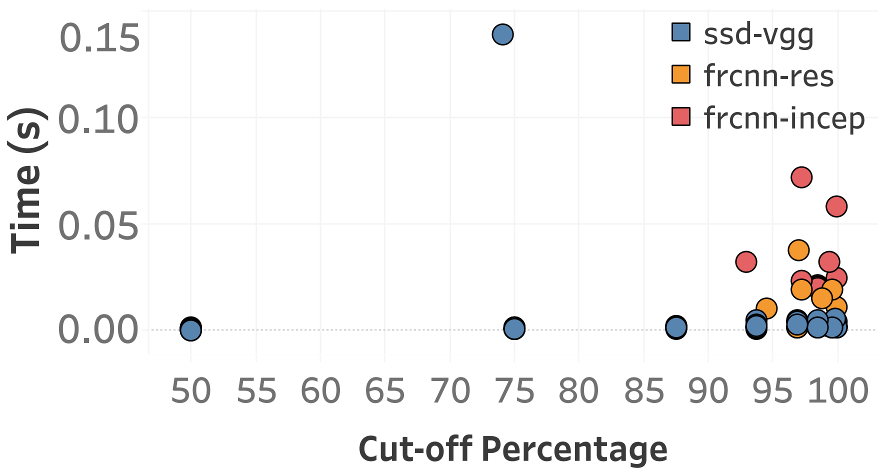


Figure 4.12: The cut-off percentages of sparse convolution units in the three models on a GPU. Each point represents the total runtime of one convolution unit.

4.5.2 End-to-end comparison

We first compare our system to running the original model on end-to-end experiments with each model and video combination. We evaluate each method (Ideal, PI controller, and machine learning) with different parameters on each video and record the average per-frame runtime and F_1 error, where $t_{F_1} = 0.8$. We use the runtimes of the original model as references. Figure 4.13 shows the results. On the y-axis, we show the relative percent savings in execution time, which is the fraction of execution time saved compared with running the original dense model. The x-axis shows the F_1 errors. Due to space constraints, we only select two representative variants for PI controller and machine learning model respectively. We discuss other variants in Section 4.5.3 and Section 4.5.4.

First, as the figure shows, Deluceva improves execution time by up to 79%, 57%, and 22% on the three models respectively with low F_1 errors below 0.1. As a reference, as we later show in Figure 4.15, the Ideal Approach improves execution time by up to 76%, 53%, and 26% on the three models (with F_1 errors equal to zero), which shows that our automated approach achieves close to the same savings as Ideal with low F_1 errors. The savings and errors vary for different video and model combinations due to their different properties. For example, videos with smaller inter-frame deltas (*e.g.*, `jn`) typically benefit from larger execution time savings, and the performance of the

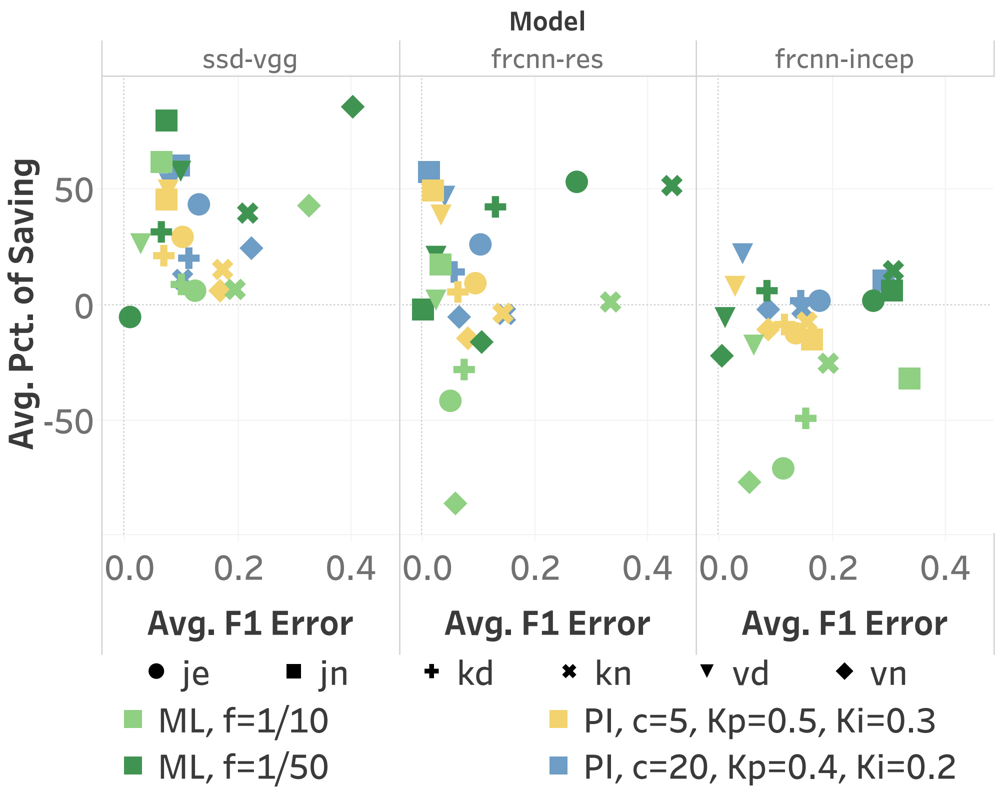


Figure 4.13: Overall performance of the PI controller and machine learning model on each model and video. Each color represents one variant. Each shape represents one video. Percentages of runtime saving and F_1 errors are shown. Y axis: percentages of runtime saving. X axis: F_1 errors. In general, variants that save more also have larger errors, and machine learning variants have either larger errors or smaller savings than PI controller variants. Overhead largely impact both methods: PI controller variants with smaller calibration frequency c have less saving and less error, same to machine learning variants with larger factor f.

largest model (`frcnn-incep`) improves least due to the overhead of convolutions with large input tensors and low-depth kernels, and the cost of `filter`, as we discussed in Section 4.5.1. For the machine-learning-based techniques, variants that save more also have larger errors due to overly high predicted target filtering percentages, while variants with smaller errors tend to choose overly-conservative target percentages. Overhead largely impacts both methods. PI controller variants with higher calibration frequency (smaller c) achieve lower savings due to the overhead, but the errors are lower as well. Machine learning model variants with larger training-to-test factor f have larger

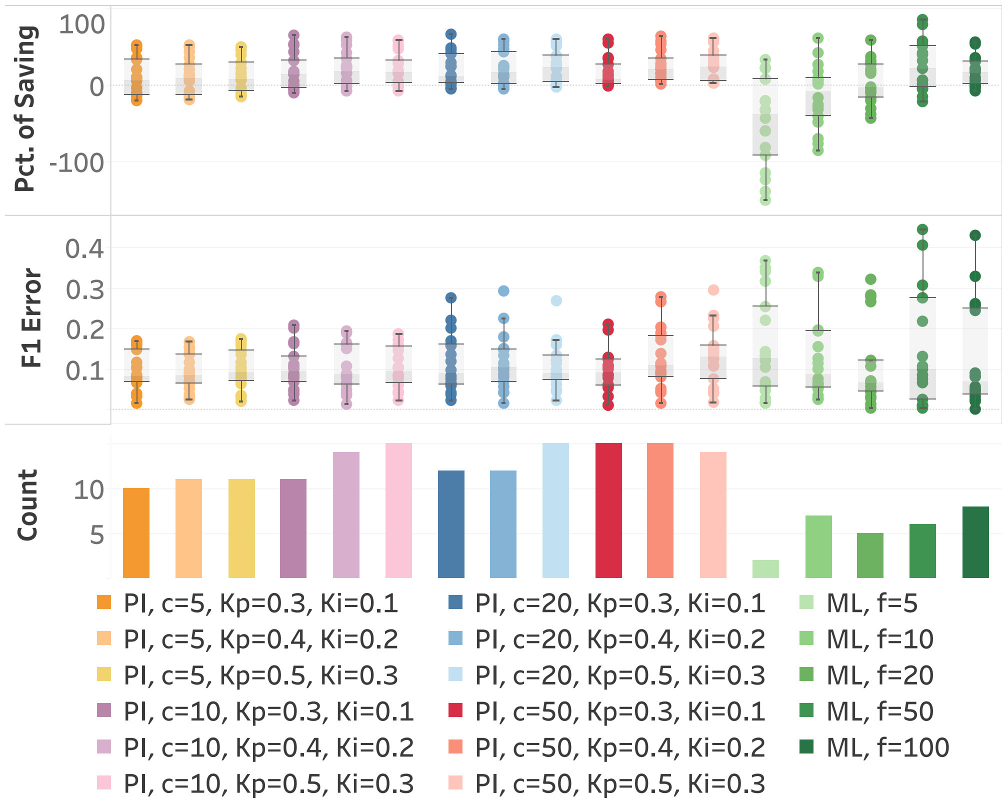


Figure 4.14: Overall performance of each method on all models and videos. Each point represents the result for one model and video combination. First row: percent savings. Second row: F_1 error. Third row: the number of (model, video) pairs that have positive savings and F_1 errors lower than 0.2. PI controller variants have larger savings, smaller F_1 errors, and lower variance compared to machine learning variants.

overheads, especially on the two larger models `frcnn-res` and `frcnn-incep`, where $f = 10$ have negative savings in many cases. Overall, PI controller variants have relatively more robust performance, similar savings and smaller errors than machine learning variants.

Next, we evaluate the robustnesses of all the variants of each method across all models and videos. We first show, in Figure 4.14, the distribution of runtime savings (first row) and F_1 errors (second row) of each variant over all models and videos. Each point represents one model variant and video combination. The results show that the machine learning model generally has higher F_1 errors with larger variance than PI controller variants, and their runtime savings are either lower than or close to the PI controller with larger variance in some cases. PI controller variants, on the other hand, have positive savings, smaller variance, and lower F_1 errors. Lower values of c lead to slightly larger overheads and lower savings, while higher values of c generate higher F_1 variances.

Another way to evaluate a method is to count the number of (model, video) pairs that have positive savings and low F_1 errors. Specifically, we filter out evaluations that have a negative saving or an F_1 error higher than 0.2 and then count the remained (model, video) pairs for each variant. The third row of Figure 4.14 shows the counts, where each color represents one variant. In general, PI controller variants have larger counts than machine learning variants. Among the PI controller variants, the ones with larger c perform better, which we think is caused by the negative savings due to larger overheads. In summary, we find that PI controller variants with a medium calibration frequency generally have good overall performance. As a representative method, a PI controller with ($c = 20$, $K_p = 0.5$, $K_i = 0.3$) has large savings (median 23%), low F_1 errors (median 0.09), low variance (standard deviation 0.06 for both saving and F_1), and large number of stable evaluations (15).

4.5.3 PI Controller

While previous experiments show that the PI controller performs well overall with a variety of parameter settings, we study the sensitivity of parameter tuning in more detail in this section. The PI controller has three parameters: the calibration frequency c, the weight of the proportional term K_p, and the weight of the integral term K_i. As above, we compare the performance of different PI controller settings to running the original model. Figure 4.15 show the average percent runtime savings and the F_1 errors for all video and model combinations. We also show the results of the Ideal Approach as reference.

Overall, we observe that the PI controller is not overly sensitive to parameter settings. We observe the greatest sensitivity for large values of c, which means infrequent calibration. In those cases, we also observe that variants that save more on runtime also have higher F_1 errors due to overly large predicted target percentages. In general, the saving of the best parameter combination is close to the saving of the Ideal Approach. In the best case, the PI controller achieves up to 65%

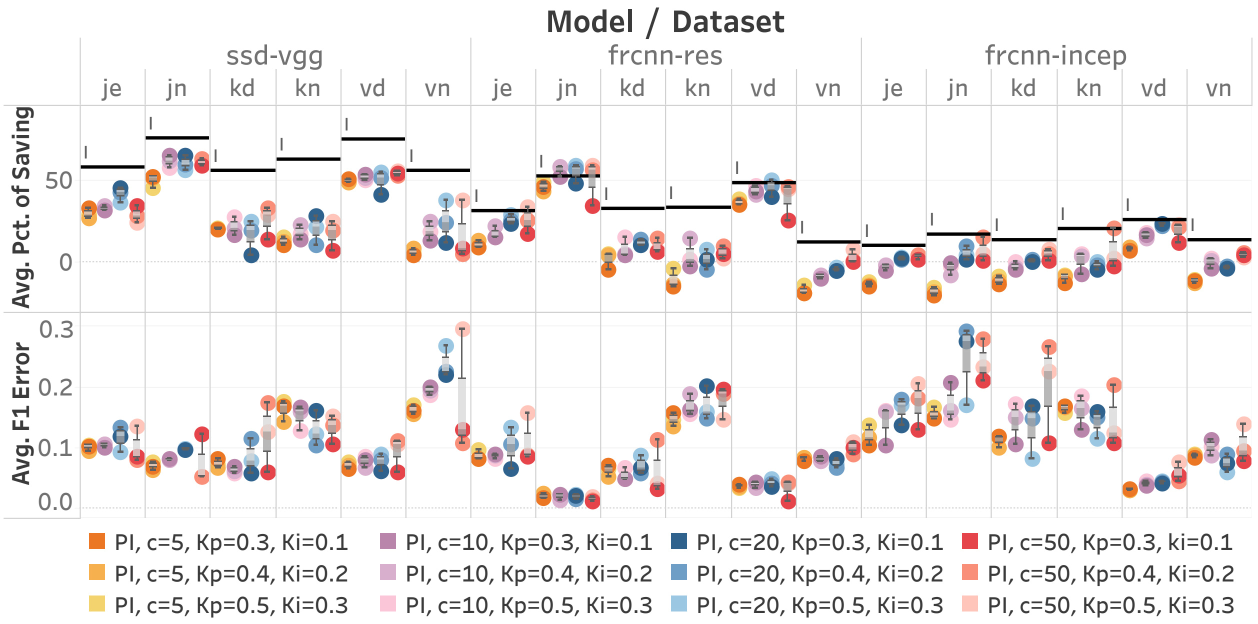


Figure 4.15: Impact of PI controller parameter values. Each color represents one combination of (c, K_p, K_i). The Ideal Approach (I) is displayed as reference lines. First row: average percentages of runtime saving. Second row: average F_1 errors. Higher frequencies (lower values of c) have less savings and lower errors and are less sensitive to K_p and K_i. The saving of the best parameter combination is close to the saving of the Ideal Approach in most cases.

runtime savings (*e.g.*, (c=20, K_p=0.3, K_i=0.1) using `ssd-vgg` on `jn`).

The calibration frequency c has a large impact on saving and error. Higher frequencies (*i.e.*, smaller cs) bring more overhead and thus reduce savings (even causing negative savings in some cases using larger models) but also lower errors as a trade-off, and the impact of different values of K_p and K_i become relatively insignificant. On the other hand, when c is large, the system is more sensitive to the values of K_p and K_i. For example, when c is 50, different combinations of K_p and K_i have savings and errors scattered in larger ranges for (`frcnn-incep`, `kd`) and (`ssd-vgg`, `vn`). We attribute this variance to the system having fewer opportunities to react to changing conditions. Due to space constraint, we only show three pairs of (K_p, K_i) for each c in Figure 4.15. We have experimented with other (K_p, K_i) pairs and observed a similar pattern with larger variances when c is large. This experiment indicates that, when c is large and the values of K_p and K_i have

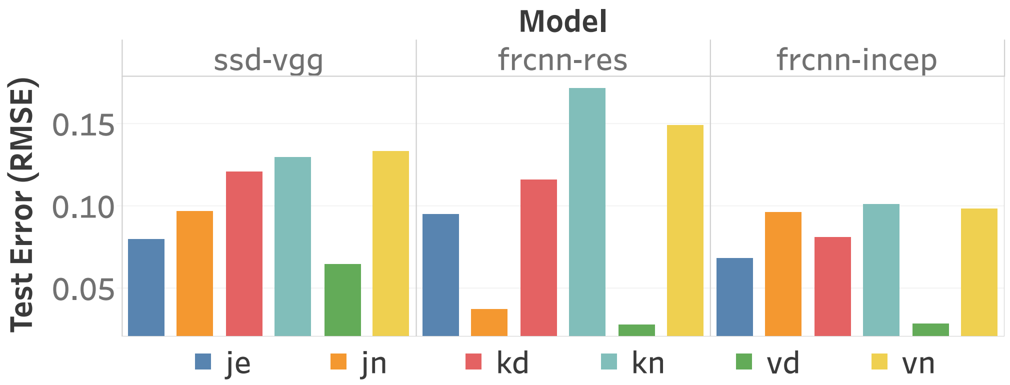


Figure 4.16: RMSEs of 10-fold cross validation of the target percentage model on the six videos using the three object detection models. All errors are lower than 0.17.

significant impact, video-and-model-specific parameter tuning is necessary if we want to find the best combination.

Importantly, for small-to-medium c values, the PI controller produces consistent results independent of the exact settings of the other two tunable parameters. We further observe that medium c values (*e.g.*, 10) achieve both low sensitivity to parameter tuning and strong performance both in terms of runtime saving and F_1 error. This is important because it means that the system can use a reasonable calibration frequency and avoid a potentially expensive parameter tuning phase.

4.5.4 Target Percentage Model

In this subsection, we first evaluate the target percentage model independently by cross validation, then we show how different training videos may affect the model prediction for a given video. Similarly, one important decision to make for using machine learning is the training-to-test factor f. We then evaluate how different values of f impact performance of the machine learning approach. Specifically, as introduced in Section 4.3.2, we divide a 300×300 frame into 5×5 overlapping blocks, each has size 100×100. The training is done using TensorFlow's `GradientDescentOptimizer`. We set the learning rate to 0.01 and run 1000 iterations. The error metric is root-mean-squared error (RMSE).

We first cross-validate the model. Figure 4.16 shows the 10-fold cross validation results of the

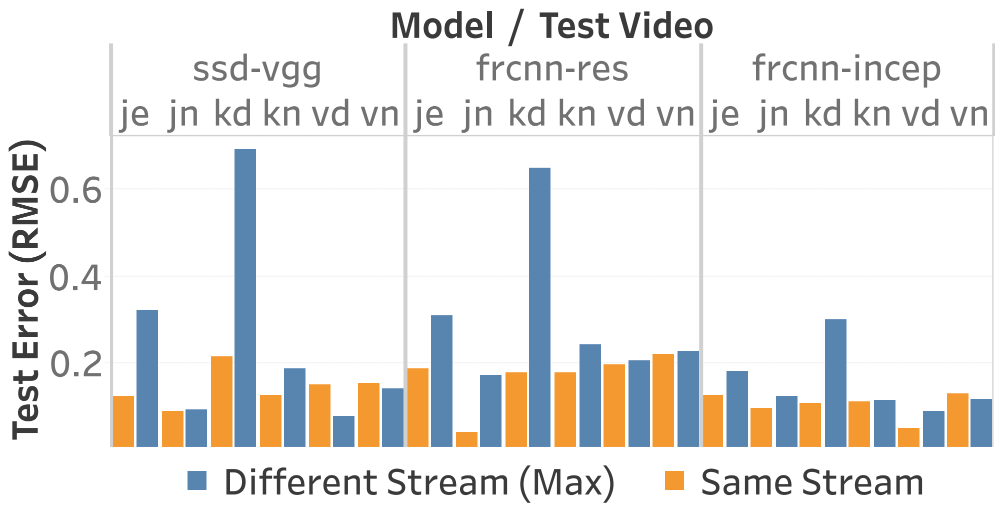


Figure 4.17: The test errors of the target percentage model trained and tested on different videos. Errors are higher when trained and tested on different videos from the same stream (orange), and much higher on videos from different streams (blue).

model on the six videos using the three object detection models. All errors are lower than 0.17, indicating that a linear model is good enough for predicting target percentages.

However, a model trained on one video may not work well on another, even when they are from the same stream. To evaluate, we train the target percentage model on one video and test it on different videos. Figure 4.17 shows the results. We distinguish training and test videos that are from the same stream but taken at different times (orange bars) and from different streams (blue bars). Among training videos from different streams, we only show the one with the largest error due to space constraints.

The models trained and tested on different videos from the same stream have higher test errors compared to Figure 4.16. If the model is trained and tested on videos from different streams, in most cases the errors are much higher, such as 0.65 when `frcnn-res` is trained on `vn` and tested on `kd`. However, there are also cases where all models have similar errors no matter which training video is used (*e.g.*, `ssd-vgg` on `jn` and `vn`). We observe that in these cases, the test video usually has a smaller number of recognizable objects, as shown in Table 4.2, where a model that constantly predicts a high filtering percentage would work well. However, it does not always indicate that the

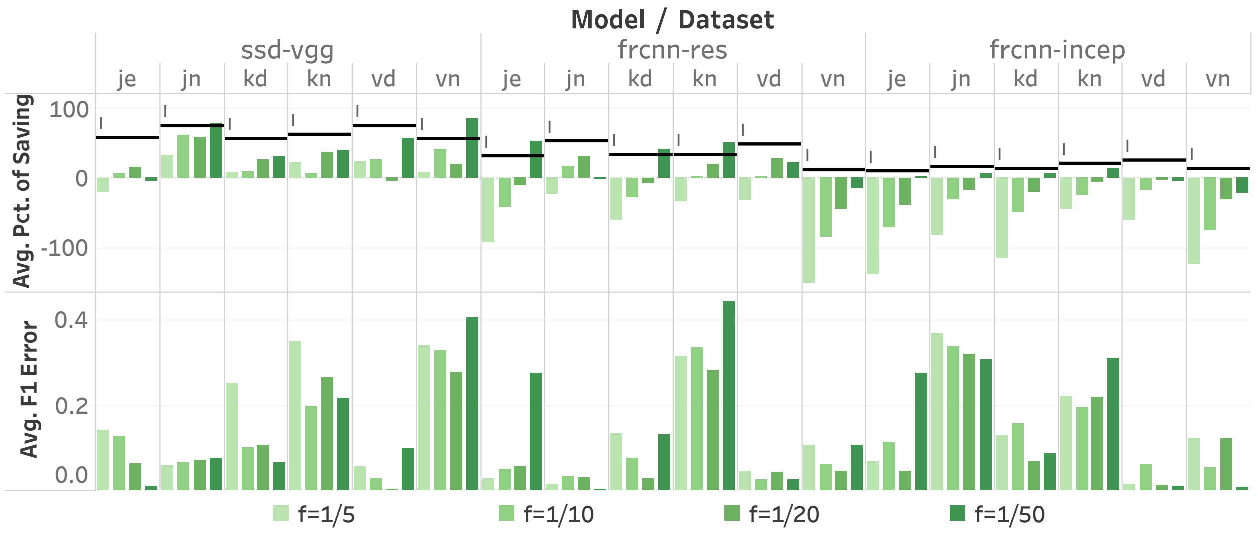


Figure 4.18: Impact of the training-to-test factor f on the target percentage model. Each color represents one value of f. The Ideal Approach (I) is displayed as reference lines. First row: average percentages of runtime savings. Second row: average F_1 errors. Larger values of f have lower savings and lower errors with smaller variances than smaller values of f.

model trained on another video predicts perfectly, since a predicted percentage larger than 100 will be adjusted to 100.

This evaluation gives us a few insights. First, model training is stream-dependent. Using a model trained on a different stream may lead to large errors. Second, even when training and testing on the same stream, larger errors still occur if the training frames and test frames have distinct properties affected by factors such as time. This suggests that retraining is necessary if the environment changes significantly.

Finally, we evaluate the impact of the training-to-test factor f. As discussed in Section 4.3.2, in order to reduce the overhead, we need to train on fewer frames and test on more frames. Figure 4.18 shows the results of evaluating the six videos using models trained on the same videos. The training frames are picked from the beginning of each video and the evaluation frames are picked from the end such that they do not overlap. Each color represents a value of f, and we also show the results of the Ideal Approach. The first row shows the percent saving on runtime, and the second row shows the F_1 errors.

As in previous experiments, greater runtime savings usually mean higher F_1 errors. When f is large, the machine learning approach has less or even negative improvement in runtime on large models. When f is small, the overhead is less but the F_1 errors are, in some cases, much larger than with small values of f (*e.g.*, `frcnn-incep` on `je`). In order to understand how f affects the predicted target percentages and thus affects savings and errors, we also investigate the values of the predictions before being truncated into the $[0, 100]$ window. We observe that when f is small (1/50 or 1/100), which means that training set is much smaller than the test set, the model tends to either overshoot (> 100) or undershoot (< 0). Smaller percentages give us more accurate results but slower runtimes and sometimes negative savings. When f is large, the predicted filtering percentages are reasonably between 0.8 and 1.0. However, the percent runtime savings are less because of the overhead. These results show that a medium f is good for the machine learning model to avoid both large overhead and instability.

4.6 Summary

Deluceva is a system that optimizes live-video analysis using deep learning by applying incremental and approximate computation techniques. Our approach includes (1) a new method for incremental deep network inference with new, specialized operators; (2) two algorithms to dynamically adjust the amount of data processed to minimize runtime subject to achieving a target quality, and (3) a new method to generating mixed networks with sparse and dense operators. Experiments on three real models and six videos show that our prototype system can achieve significant performance gains up to 79% with F_1 errors below 0.1.

Chapter 5
RELATED WORK

In this chapter, we cover the literature of large-scale data analytics, especially relevant work in the context of the following directions.

5.1 Related Work on Recursive Datalog Evaluation Engines

Adaptive query processing: The Eddy query processing mechanism [48] dynamically reorders operators in a query plan. It detects places where the reordering can happen, and routes tuples iteratively through operators based on costs. In contrast, our work focuses on query plans that are static but have loops.

Iterative MapReduce: MapReduce [71] and Hadoop [221] are known to be inefficient for iterative applications and several systems have been developed to address this limitation including HaLoop [55] and Twister [75]. Besides supporting iterations, PrIter [227] also provides the ability to prioritize the execution of subsets of data. OptIQ [163] uses program analysis to detect loop-variant data and evaluate it incrementally. [41] observes that recursive tasks only deliver output at the end and thus increases the cost of fault-tolerance. In contrast to our work, systems that extend Hadoop can only support synchronous iterations.

Synchronous-only systems: Beyond MapReduce, multiple systems have been designed for iterative applications and have introduced their own programming models. Some of them focus on *graph applications*, while others have more general programming models. In Pregel [152], a program consists of iterations, and in each iteration vertices can receive messages from the previous iteration, update states, and send messages out. Pregelix [56], which is built on top of Hyracks [52], is similar to Pregel but also supports both in-memory and out-of-core workloads efficiently. GraphX [91] is a graph processing framework built on top of Spark [225], a distributed in-memory dataflow engine.

REX [157] provides a programming model that focuses on delivering deltas across iterations. All these systems focus on synchronous iterative computations.

Systems that also support asynchronous iterations: These systems generally have programming models that are based on message passing between units, such as graph vertices. They typically provide a set of low-level interfaces for users to implement their own applications. Some of these systems are specialized for graph processing, such as GraphLab [146] and Grace [212], while others are more general. Stratosphere [77] focuses on incremental iteration evaluation with different granularities: *superstep* for a full iteration and *microstep* for a single tuple. Naiad [159] proposes a general-purpose dataflow framework that supports nested loops. epiC [122] adopts the Actor-like programming model to encapsulate various parallel processing models into one system. To choose between a synchronous and asynchronous model, PowerSwitch [223] does the first comparison and comes up with a cost model to guide the switch between the two models. In contrast, our approach generates query plans from Datalog programs and these plan require only small changes to an existing shared-nothing system.

Datalog evaluation systems: Several systems focus on evaluating Datalog (with extensions) or equivalent high-level declarative languages. LogicBlox [21] is a single-machine commercial system that focuses on Datalog evaluation. In contrast, we focus on a shared-nothing implementation. GLog [84] provides a language similar to Datalog with extensions, then translates such a program into MapReduce jobs, which support only synchronous iterations. SociaLite [179, 180] is a distributed system that evaluates Datalog programs with meet aggregate functions. Asynchronous execution is supported within each *epoch* but not for the entire program. More importantly, the implementation is based on code-generation instead of having a general-purpose query engine. DeALS [8] is a research project about Datalog evaluation with support for aggregate functions in recursion with sequential [186], single-machine multi-core [224], and cluster (on Spark [225]) [185] implementations. Theoretically, the semantics of monotonic aggregation in recursion is investigated in Ross et al. [173] and DatalogFS [153]. CALM [43] is a set of principles that connect distributed consistency with logical monotonicity, which leads to the Bloom [43] language. Bloom helps users identify unnecessary coordination. Bloom^L [64] is an extension to Bloom [43] with lattices, but without support for asynchronous evaluation on them.

Iterations in scientific workflows: The Orbit [70] operator provides support for the iterative processing of workflow fragments. In contrast, our work focuses on Datalog programs.

Failure handling: Discussed in Section 2.3.

5.2 *Related Work on Memory Management for Cloud Data Analytics*

Memory allocation within a single machine: Many approaches focus on sharing memory across multiple objects on a single machine. Several techniques have *queries* as the objects: Some [60, 78, 162] allocate buffer space across queries based on page access models to reduce page faults. Others [53, 165] tune buffer allocation policies to meet performance goals in real-time database systems. A third set of methods [201] uses application resource sensitivities to guide allocation. More recently, Narasayya *et al.* [161] develop techniques to share a bufferpool across multiple *tenants*. Several approaches focus on *operators* within a query. Anciaux *et al.* [44] allocate memory across operators on memory-constrained devices. Davison *et al.* [69] sell resources to competing operators to maximize profit. Garofalakis *et al.* [85] schedule operators with multidimensional resource constraints in NUMA systems. Finally, Storm *et al.* [193] manage memory across *database system components*. Although they share the idea of managing memory for multiple objects with a global objective function, the problems are restricted to single machines, and they ignore GC. Salomie *et al.* [176] move memory across JVMs dynamically by adding a balloon space to OpenJDK but have no performance models or scheduling algorithms. Ginkgo [110] dynamically manages memory for multiple Java applications by changing layouts using Java Native Interface. However, it models performance by profiling specific workloads, while our approach is applicable to arbitrary relational queries.

Cluster-wide resource scheduling: Some techniques develop models to understand how resources affect the runtime characteristics of applications. Li *et al.* [140] partition queries on heterogeneous machines based on system calibrations and optimizer statistics. Herodotou *et al.* [107, 108] tune Hadoop application parameters based on machine learning models built by job profiles. Some other techniques focus on *short-lived requests*. Lang *et al.* [135] schedule transactional workloads on heterogeneous hardware resources for multiple tenants. Schaffner *et al.* [178] minimize tail latency of tenant response times in column database clusters. BlowFish [127] adaptively adjusts storage for performance of random access and search queries by switching between array layers with different sampling rates based on certain thresholds. In contrast, our focus is relational queries on Java-based systems with no sampling. To provide a unified framework for resource sharing and application scheduling, several *general-purpose* resource managers have emerged [109, 207, 218]. However, they all lack the ability to adjust memory limits dynamically.

Adaptive GC tuning: Cook *et al.* [65] provide two GC triggering policies based on real-time statistics, but do not investigate memory management across applications. Simo *et al.* [187] study the performance impact of JVM heap growth policies by evaluating them on several benchmarks. Maas *et al.* [151] observe that GC coordination is important for distributed applications. They let users specify coordination policy to make all JVMs trigger GC at the same time under certain conditions.

Region-based memory management: Another line of work uses region-based memory management (RBMM) [202] to avoid GC overhead. Broom [90] categorizes Naiad [159] objects into three types with a region assigned to each. Deca [149] manipulates Spark Scala objects in-memory representations as byte arrays and allocates pages for them. While RBMM may reduce GC overhead, it requires that the programmer declare object-to-region mappings and adds complexity to compilation, without eliminating space safety concerns [100].

5.3 Related Work on Neural Network Inference for Video Analytics

Convolutional neural network (CNN) architectures: Although many vision models have complex structures, they are mostly built on top of their predecessors [132, 189] with simple architectures of stacks of common layers such as convolution and pooling, which have been proven to work well on vision tasks [15, 131, 142]. Some techniques study architectural extremes, such as very deep [188], shallow [49], or fully convolutional [182] networks. More recently, people have discovered micro structures that can significantly benefit accuracy [106, 119, 198, 199]. Another trend is to develop simple, energy-efficient models that can be deployed on mobile devices [112, 170].

Object detection models: An object detection model generates two outputs: object bounding boxes and class labels. Although they have different architectures, they usually use an established CNN as the raw image feature extractor then perform localization and classification using extracted features. Different base networks bring different accuracies and runtimes to each architecture [114]. Some [88, 89, 172, 181] treat localization and classification as two steps while others [68, 144, 171] combine them into one for fast inference.

Vision tasks on videos: For *object detection* on videos, most approaches apply image object detection models to each frame [105, 124, 125, 171]. [192] uses Long Short Term Memory (LSTM) to remember previously detected but partially occluded human faces. For *video classification*, [126] proposes several architectures that fuse temporal features at different levels. *Object tracking* is another common video analysis task, where object bounding boxes are produced for each frame given initial boxes. Most state-of-the-art techniques use models based on CNNs. For example, in the Visual Object Tracking challenge 2017 [130], most top performers [79, 92, 160, 195] use CNNs.

Model sparsity: Many people have worked on reducing the size of models. [57, 111] distill an ensemble of models into a smaller model without loss of accuracy. Others prune network connections directly by various techniques, such as training [104], Taylor expansion [158], weight hashing and sharing [61], Huffman encoding [103], and circulate matrices [63]. XNOR-Net [170] creates models with binary weights and operators for fast evaluation and memory saving. In contrast to these work where a pruned model is used for inference with no more change, our system takes a pre-trained model and prunes it dynamically during model evaluation based on error feedback.

Model cascade: There is always a trade-off between accuracy and speed. *Model cascade* refers to a line of techniques that combines a sequence of classifiers/models with different properties and exit early when possible to improve inference speed. As representative early work, [209] cascades traditional image processing features by short-circuiting classifiers with confident outputs. Some techniques focus on specific tasks on images, such as pedestrian detection [58], human faces detection [139], and facial key point detection [196]. More recently, Shen et al. [183] and NoScope [124] cascades models from universal ones to specialized ones to avoid using overkilling, "oracle" models. These approaches require offline training and hyperparameter tuning for a given reference model, while our approach can take any reference model off-the-shelf and deploy it directly without much offline overhead and is adaptive to dynamic input. Moreover, our approach can be applied on top of these specialized models to further reduce their inference times.

Video analytics and management systems: Several video database systems have been proposed in early work [45] and also recently [150]. These systems offer traditional database system features such as declarative SQL-like languages, indexing, image and semantic-based querying, storage management, and query optimization. Focus [113] is a system for querying objects of certain classes (*e.g.*, cars) from many days of recorded video with low latency and low cost. VideoStorm [226] is a video analytics system for live video analysis over large clusters. It focuses on resource management for video queries to maximize performance on quality and lag. In contrast to these systems, we focus on accelerating video analytics task by incremental and approximate processing.

Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

Large-scale data analytics is key to modern science, technology, and business development. Big data systems have emerged rapidly in recent years, but modern data analytics still remains challenging due to application requirements: users need to analyze vasts amount of data generated from various sources; analysis efficiency is critical in many scenarios; most big data systems are built on top of new operating environments, such as clusters and cloud services, where resource management is important; high-level, feature-rich programming interfaces are required to support a high variety of data and workload types.

In this dissertation, we presented methods to improve system efficiency for large-scale data analytics. We investigated the problem in the context of the following three research projects addressing the same key problem, *optimization of analytical query execution*, in different ways. Specifically, these projects focused on *runtime* optimzation, which considers not only *static* information that is available prior to the actual execution, but more importantly, runtime information. We demonstrated, from these projects, that runtime optimzation can significantly improve overall system performance in terms of runtime, resource utilization, and application failure.

We first presented a full-stack solution for large-scale recursive relational query evaluation in shared-nothing engines (Chapter 2). With our approach, users can express their analysis using a high-level declarative language (a subset of Datalog with aggregate functions). Queries are then compiled into distributed query plans with termination guarantee and support for multiple execution models, including synchronous, asynchronous, and prioritized processing. Our approach can be applied to any existing shared-nothing engine with small extensions. We implemented our approach in Myria and empirically evaluated the interplay between execution model, failure handling method, and application property. We found that no single iterative execution strategy outperforms all others;

rather, application properties must drive method selection.

Next, we presented ElasticMem, an approach for automatic and elastic memory management of cloud data analytics applications (Chapter 3). In clouds or clusters, a resource manager schedules applications to containers with hard memory limits, which requires accurate application memory usage estimation before launching containers. However, memory estimation for large-scale analytical applications is difficult, and inappropriate estimate can lead to failures and performance degredation. ElasticMem avoids pre-execution memory usage estimation by elastically allocating memory across containers during runtime. It includes a technique to dynamically change JVM memory limits, an approach to model memory usage and garbage collection cost during query execution, and a memory manager that performs actions on JVMs to reduce total failures and run times. Our evaluation of the approach on our prototype implementation showed that ElasticMem outperformed static memory allocation in terms of query failure, garbage collection time, query execution time, and overall resource utilization.

Lastly, we presented Deluceva, a system that dynamically optimizes live video analysis using neural network models by applying incremental and approximate computation techniques (Chapter 4). Many video analysis approaches apply image neural network models directly on each video frame. While being easy to develop, they do not consider the rich temporal redundancy of videos. Deluceva accelerates model inference by dynamically selecting sufficiently significant temporal deltas to process for each video frame. It includes a new method for incremental deep network inference with specialized operators, two algorithms to dynamically adjust the granularity of deltas towards a target quality, and a method to generate mixed networks with sparse and dense operators. The evaluation using our prototype implementation and multiple models and videos as input showed that Deluceva can significantly improve performance while keeping object detection errors low.

Several interesting research directions remain open for each of the above three projects.

In the problem of recursive query evaluation, an important direction of future work is to develop a cost-based optimizer to select the least-cost plan for each application, including choosing between synchronous and asynchronous execution and selecting the pull order for each join operator. Plan selection can happen before execution, which is similar to traditional static query optimization.

Moreover, cost-based plan adjustment during runtime (*e.g.*, an Eddy-style [48] executor) based on the observed statistics is a more interesting and challenging direction.

In the context of ElasticMem, our primary focus are relational queries running in Java-based containers and having hash tables as the major large in-memory data structures. Extending our approach to other data structures, more diverse workloads, and more systems and runtimes is interesting future work. Another direction is to investigate ElasticMem's performance on large-scale cluster deployments, *e.g.*, hundreds of thousands of applications and machines.

For Deluceva, our implementation for sparse convolution operator does not target any specific optimizations besides vectorization. An interesting future direction is to further optimize its performance by taking memory hierarchy and blocking techniques into consideration. Additional optimizations are also possible for the filter operator. Another important direction is to evaluate Deluceva on GPUs and on other neural network models besides object detction models. Finally, we want to investigate its performance on mobile and edge devices, where enery consumption, memory bandwidth, and runtime are all important factors.

BIBLIOGRAPHY

[1] Amazon EC2 on-demand instances pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[2] Amazon EC2 spot instances. http://aws.amazon.com/ec2/purchasing-options/spot-instances/.

[3] Amazon Redshift. https://aws.amazon.com/redshift/.

[4] Amazon web service. https://aws.amazon.com/.

[5] Apache flink. http://flink.apache.org/.

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

[7] Common table expression. https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL#Common_table_expression.

[8] Deductive application language system (deals). http://wis.cs.ucla.edu/deals/.

[9] Docker Container. https://www.docker.com/.

[10] Greenplum. http://pivotal.io/big-data/pivotal-greenplum-database.

[11] The Hadoop ecosystem table. http://hadoopecosystemtable.github.io/.

[12] How many CCTV cameras actually are there in the UK? http://www.arc24.co.uk/how-many-cctv-cameras-are-actually-are-there-in-the-uk/.

[13] How we built Uber engineerings highest query per second service using Go. https://eng.uber.com/go-geofence/.

[14] `im2col`. https://www.mathworks.com/help/images/ref/im2col.html.

[15] ImageNet. http://image-net.org/index.

[16] Internet of things. https://en.wikipedia.org/wiki/Internet_of_things.

[17] Jackson Hole live stream. https://www.youtube.com/watch?v=a7jwCYuQarU.

[18] Knapsack problem. https://en.wikipedia.org/wiki/Knapsack_problem.

[19] Kubernetes. http://kubernetes.io/.

[20] Kusatsu live stream. https://www.youtube.com/watch?v=a17vvYJZP0w.

[21] LogicBlox inc. http://www.logicblox.com/.

[22] Mass surveillance in China. https://en.wikipedia.org/wiki/Mass_surveillance_in_China.

[23] Myria: Big Data as a Service. http://myria.cs.washington.edu/.

[24] The PASCAL visual object classes. http://host.robots.ox.ac.uk/pascal/VOC/.

[25] PID controller. https://en.wikipedia.org/wiki/PID_controller.

[26] The prevalence of motor vehicle crashes involving road debris, united states, 2011-2014. http://exchange.aaa.com/wp-content/uploads/2017/05/RoadDebris_FACTSHEET-1.pdf.

[27] SDSS SkyServer DR7. http://skyserver.sdss.org/dr7.

[28] Single shot multibox detector in tensorflow. https://github.com/balancap/SSD-Tensorflow.

[29] TPC-H. http://www.tpc.org/tpch/.

[30] TPC-H queries in MyriaL. https://github.com/uwescience/tpch-myrial.

[31] Traffic safety facts 2015. https://crashstats.nhtsa.dot.gov/Api/Public/Publication/812384.

[32] Tungsten: memory management and binary processing on Spark. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[33] Venice live stream. https://www.youtube.com/watch?v=vPbQcM4k1Ys.

[34] Vertica. https://www.vertica.com/.

[35] Vgg16. http://www.cs.toronto.edu/~frossard/post/vgg16/.

[36] Walmart: Big data analytics at the worlds biggest retailer. https://www.bernardmarr.com/default.asp?contentID=690.

[37] YouTube statistics. https://www.youtube.com/intl/en-GB/yt/about/press/.

[38] Memory management in the Java HotSpot™virtual machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, 2006.

[39] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[40] Azza Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[41] Foto N. Afrati et al. Mapreduce extensions and recursive queries. In *EDBT*, 2011.

[42] Sattam Alsubaiee et al. AsterixDB: A scalable, open source BDMS. In *VLDB*, 2014.

[43] Peter Alvaro et al. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[44] Nicolas Anciaux, Luc Bouganim, and Philippe Pucheral. Memory requirements for query execution in highly constrained devices. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 694–705. VLDB Endowment, 2003.

[45] Walid G. Aref, Ann C. Catlin, Jianping Fan, Ahmed K. Elmagarmid, Moustafa A. Hammad, Ihab F. Ilyas, Mirette S. Marzouk, and Xingquan Zhu. A video database management system for advancing video database research. In *In Proc. of the Int Workshop on Management Information Systems. Nov*, pages 8–17, 2002.

[46] Remzi H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. Comput. Syst.*, 21(1):36–86, February 2003.

[47] Remzi H. Arpaci-Dusseau et al. Cluster I/O with river: Making the fast case common. In *IOPADS*, 1999.

[48] Ron Avnur et al. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[49] Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2654–2662, Cambridge, MA, USA, 2014. MIT Press.

[50] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *The Journal of Logic Programming*, 4(3):259 – 262, 1987.

[51] Stephen Blackburn, Richard E. Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 153–164, 2002.

[52] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[53] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 328–341, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[54] Paul G. Brown. Overview of scidb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[55] Y. Bu et al. HaLoop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[56] Yingyi Bu et al. Pregelix: Big(ger) graph analytics on a dataflow engine. In *VLDB*, 2014.

[57] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. ACM.

[58] Zhaowei Cai, Mohammad J. Saberian, and Nuno Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. *CoRR*, abs/1507.05348, 2015.

[59] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007.

[60] Chung-Min Chen and Nick Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 342–353, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[61] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing convolutional neural networks in the frequency domain. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1475–1484, New York, NY, USA, 2016. ACM.

[62] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 125–136, 2001.

[63] Yu Cheng, Felix X. Yu, Rogério Schmidt Feris, Sanjiv Kumar, Alok N. Choudhary, and Shih-Fu Chang. Fast neural networks with circulant projections. *CoRR*, abs/1502.03436, 2015.

[64] Neil Conway et al. Logic and lattices for distributed programming. In *SoCC*, 2012.

[65] Jonathan E. Cook, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. Semi-automatic, self-adaptive control of garbage collection rates in object databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 377–388, New York, NY, USA, 1996. ACM.

[66] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 16, pages 425–427. The MIT Press, 3nd edition, 2009.

[67] C. Creusot and A. Munawar. Real-time small obstacle detection on highways using compressive rbm road reconstruction. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 162–167, June 2015.

[68] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 379–387. Curran Associates, Inc., 2016.

[69] Diane L. Davison and Goetz Graefe. Dynamic resource brokering for multi-user query execution. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 281–292, New York, NY, USA, 1995. ACM.

[70] Douglas Ericson M. de Oliveira et al. Orbit: Efficient processing of iterations. In *SBBD*, 2013.

[71] Jeffrey Dean et al. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[72] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[73] Jack Dongarra, Andrew Lumsdaine, Xinhiu Niu, Roldan Pozo, and Karin Remington. Lapack working note 74: A sparse matrix library in c++ for high performance architectures. Technical report, Knoxville, TN, USA, 1994.

[74] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, June 2002.

[75] Jaliya Ekanayake et al. Twister: A runtime for iterative MapReduce. In *HPDC*, 2010.

[76] Martin Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, 1996.

[77] S. Ewen et al. Spinning fast iterative data flows. In *VLDB*, 2012.

[78] Christos Faloutsos, Raymond T. Ng, and Timos K. Sellis. Predictive load control for flexible buffer allocation. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 265–274, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[79] Heng Fan and Haibin Ling. Sanet: Structure-aware network for visual tracking. *CoRR*, abs/1611.06878, 2016.

[80] Christoph Feichtenhofer, Axel Pinz, and Richard Wildes. Spatiotemporal residual networks for video action recognition. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3468–3476, 2016.

[81] Christoph Feichtenhofer, Axel Pinz, and Richard P Wildes. Spatiotemporal multiplier networks for video action recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[82] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(9):1627–1645, September 2010.

[83] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 596–609, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[84] Jun Gao et al. GLog: A high level graph analysis system using MapReduce. In *ICDE*, 2014.

[85] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 296–305, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[86] Roman Geus and Stefan Röllin. Towards a fast parallel sparse symmetric matrix-vector multiplication. *Parallel Comput.*, 27(7):883–896, May 2001.

[87] Ahmad Ghazal et al. Adaptive optimizations of recursive queries in teradata. In *SIGMOD*, 2012.

[88] R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, Dec 2015.

[89] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Region-based convolutional networks for accurate object detection and segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(1):142–158, Jan 2016.

[90] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[91] Joseph E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[92] Daniel Gordon, Ali Farhadi, and Dieter Fox. Re3 : Real-time recurrent regression networks for object tracking. *CoRR*, abs/1705.06368, 2017.

[93] Daniel Gordon, Ali Farhadi, and Dieter Fox. Re3 : Real-time recurrent regression networks for object tracking. *CoRR*, abs/1705.06368, 2017.

[94] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[95] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. In-memory performance for big data. *Proc. VLDB Endow.*, 8(1):37–48, September 2014.

[96] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Found. Trends databases*, 5(2):105–195, November 2013.

[97] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[98] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM.

[99] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[100] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM.

[101] Daniel Halperin. Simplifying the configuration of 802.11 wireless networks with effective snr. *CoRR*, abs/1301.6644, 2012.

[102] Daniel Halperin et al. Demo of the Myria big data management service. In *SIGMOD*, 2014.

[103] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[104] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[105] Wei Han, Pooya Khorrami, Tom Le Paine, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, and Thomas S. Huang. Seq-nms for video object detection. *CoRR*, abs/1602.08465, 2016.

[106] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[107] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[108] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272, 2011.

[109] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[110] Michael R. Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Ryu, and Muli Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 130–137, Washington, DC, USA, 2011. IEEE Computer Society.

[111] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[112] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[113] K. Hsieh, G. Ananthanarayanan, P. Bodik, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. *ArXiv e-prints*, January 2018.

[114] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[115] J. Hwang et al. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[116] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, Asilomar, California, 2007.

[117] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.

[118] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 268–277, New York, NY, USA, 1991. ACM.

[119] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[120] Michael Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[121] Joo Seong Jeong, Woo-Yeon Lee, Yunseong Lee, Youngseok Yang, Brian Cho, and Byung-Gon Chun. Elastic memory: Bring elasticity back to in-memory big data analytics. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 3–3, Berkeley, CA, USA, 2015. USENIX Association.

[122] Dawei Jiang et al. epiC: An extensible and scalable system for processing big data. In *VLDB*, 2014.

[123] Bo Kågström, Per Ling, and Charles van Loan. Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, September 1998.

[124] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. NoScope: Optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, August 2017.

[125] Kai Kang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Object detection from video tubelets with convolutional neural networks. *CoRR*, abs/1604.04053, 2016.

[126] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 1725–1732, Washington, DC, USA, 2014. IEEE Computer Society.

[127] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association.

[128] Nodira Khoussainova. Leveraging usage history to enhance database usability. 2012.

[129] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[130] Matej Kristan, Ales Leonardis, Jiri Matas, Michael Felsberg, Roman Pflugfelder, Luka Cehovin Zajc, Tomas Vojir, Gustav Hager, Alan Lukezic, Abdelrahman Eldesokey, and Gustavo Fernandez. The visual object tracking vot2017 challenge results. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[131] Matej Kristan, Jiri Matas, Aleš Leonardis, Tomas Vojir, Roman Pflugfelder, Gustavo Fernandez, Georg Nebehay, Fatih Porikli, and Luka Čehovin. A novel performance evaluation methodology for single-target trackers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(11):2137–2155, Nov 2016.

[132] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[133] Haewoon Kwak et al. What is Twitter, a social network or a news media? In *WWW*, 2010.

[134] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

[135] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. Towards multi-tenant performance slos. *IEEE Trans. on Knowl. and Data Eng.*, 26(6):1447–1463, June 2014.

[136] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *International Conference on Parallel Processing, 2004. ICPP 2004.*, pages 169–176 vol.1, Aug 2004.

[137] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[138] Jure Leskovec et al. Snap.py: SNAP for Python, a general purpose network analysis and graph mining tool in Python. `http://snap.stanford.edu/snappy`, 2014.

[139] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.

[140] Jiexing Li, Jeffrey Naughton, and Rimma V. Nehme. Resource bricolage for parallel database systems. *Proc. VLDB Endow.*, 8(1):25–36, September 2014.

[141] Leonid Libkin and Limsoon Wong. On representation and querying incomplete information in databases with multisets, 1995.

[142] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. *Microsoft COCO: Common Objects in Context*, pages 740–755. Springer International Publishing, Cham, 2014.

[143] Bin Liu, Yali Zhu, and Elke Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 347–358, New York, NY, USA, 2006. ACM.

[144] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. *SSD: Single Shot MultiBox Detector*, pages 21–37. Springer International Publishing, Cham, 2016.

[145] Sarah Loebman et al. Big-data management use-case: A cloud service for creating and analyzing galactic merger trees. DanaC, 2014.

[146] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[147] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–, Washington, DC, USA, 1999. IEEE Computer Society.

[148] Large Synoptic Survey Telescope. `http://www.lsst.org/`.

[149] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9(12):936–947, August 2016.

[150] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 57–70, New York, NY, USA, 2016. ACM.

[151] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[152] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[153] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *The VLDB Journal*, 22(4):471–493, August 2013.

[154] Joseph McKendrick. The petabyte challenge: 2011 ioug database growth survey.

[155] Frank McSherry et al. Differential dataflow. In *CIDR*, 2013.

[156] H. Menon et al. Adaptive Techniques for Clustered N-Body Cosmological Simulations. *ArXiv e-prints*, September 2014.

[157] S.R. Mihaylov et al. REX: Recursive, delta-based data-centric computation. In *VLDB*, 2012.

[158] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

[159] Derek G. Murray et al. Naiad: A timely dataflow system. In *SOSP*, 2013.

[160] Hyeonseob Nam and Bohyung Han. Learning multi-domain convolutional neural networks for visual tracking. *CoRR*, abs/1510.07945, 2015.

[161] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, February 2015.

[162] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, SIGMOD '91, pages 387–396, New York, NY, USA, 1991. ACM.

[163] Makoto Onizuka et al. Optimization for iterative queries on MapReduce. In *VLDB*, 2013.

[164] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 129–140, 2002.

[165] Hwee Hwa Pang, Michael J. Carey, and Miron Livny. Managing memory for real-time queries. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 221–232, New York, NY, USA, 1994. ACM.

[166] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[167] Ross J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, 1992. World Scientific.

[168] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *Proceedings of the 1994 International Symposium on Logic Programming*, ILPS '94, pages 204–218, Cambridge, MA, USA, 1994. MIT Press.

[169] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the coral deductive database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 167–176, New York, NY, USA, 1993. ACM.

[170] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[171] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[172] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.

[173] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 114–126, New York, NY, USA, 1992. ACM.

[174] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. ImageNet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[175] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[176] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA, 2013. ACM.

[177] Mohamed Sarwat et al. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. In *VLDB*, 2013.

[178] Jan Schaffner, Benjamin Eckart, Dean Jacobs, Christian Schwarz, Hasso Plattner, and Alexander Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, pages 1264–1275. IEEE Computer Society, 2011.

[179] Jiwon Seo et al. Distributed SociaLite: A Datalog-based language for large-scale graph analysis. In *VLDB*, 2013.

[180] Jiwon Seo et al. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, 2013.

[181] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *CoRR*, abs/1312.6229, 2013.

[182] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1605.06211, 2016.

[183] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2197–2205, 2017.

[184] Yanyan Shen et al. Fast failure recovery in distributed graph processing systems. In *VLDB*, 2014.

[185] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1135–1149, New York, NY, USA, 2016. ACM.

[186] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in deals. *2015 IEEE 31st International Conference on Data Engineering*, pages 867–878, 2015.

[187] José Simão and Luís Veiga. Adaptability driven by quality of execution in high level virtual machines for shared cloud environments. *Comput. Syst. Sci. Eng.*, 28(6), 2013.

[188] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[189] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[190] Sloan Digital Sky Survey. http://cas.sdss.org/.

[191] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[192] Russell Stewart and Mykhaylo Andriluka. End-to-end people detection in crowded scenes. *CoRR*, abs/1506.04878, 2015.

[193] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.

[194] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Trans. Cir. and Sys. for Video Technol.*, 22(12):1649–1668, December 2012.

[195] Chong Sun, Huchuan Lu, and Ming-Hsuan Yang. Learning spatial-aware regressions for visual tracking. *CoRR*, abs/1706.07457, 2017.

[196] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deep convolutional network cascade for facial point detection. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 3476–3483, Washington, DC, USA, 2013. IEEE Computer Society.

[197] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, Inception-ResNet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[198] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[199] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[200] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 578–587, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[201] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[202] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, February 1997.

[203] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013.

[204] University of Washington eScience Institute. http://escience.washington.edu/.

[205] Prasang Upadhyaya et al. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD*, 2011.

[206] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, January 2005.

[207] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[208] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, 1986.

[209] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–511–I–518 vol.1, 2001.

[210] W. Vogels et al. The design and architecture of the Microsoft Cluster Service - a practical approach to high-availability and scalability. In *FTCS*, 1998.

[211] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[212] Guozhang Wang et al. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[213] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The Myria big data management and analytics system and cloud services. In *CIDR*, 2017.

[214] Jingjing Wang and Magdalena Balazinska. Toward elastic memory management for cloud data analytics. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, pages 7:1–7:4, New York, NY, USA, 2016. ACM.

[215] Jingjing Wang and Magdalena Balazinska. Elastic memory management for cloud data analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 745–758, Santa Clara, CA, 2017. USENIX Association.

[216] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. VLDB Endow.*, 8(12):1542–1553, August 2015.

[217] Y. Wang and I. H. Witten. Induction of model trees for predicting continuous classes. In *Poster papers of the 9th European Conference on Machine Learning*. Springer, 1997.

[218] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Russel Sears, Beysim Sezgin, and Julia Wang. Reef: Retainable evaluator execution framework. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1343–1355, New York, NY, USA, 2015. ACM.

[219] Philippe Weinzaepfel, Zaïd Harchaoui, and Cordelia Schmid. Learning to track for spatio-temporal action localization. *CoRR*, abs/1506.01929, 2015.

[220] Seth J. White and David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 419–431, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[221] Tom White. *Hadoop: The Definitive Guide*. 2009.

[222] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007.

[223] Chenning Xie et al. SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In *PPoPP*, 2015.

[224] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. Parallel bottom-up evaluation of logic programs: Deals on shared-memory multicore machines. In *ICLP*, 2015.

[225] Matei Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[226] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 377–392, Berkeley, CA, USA, 2017. USENIX Association.

[227] Y. Zhang et al. PrIter: a distributed framework for prioritized iterative computations. In *SoCC*, 2011.

[228] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. Fast human detection using a cascade of histograms of oriented gradients. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 1491–1498, Washington, DC, USA, 2006. IEEE Computer Society.

[229] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.