Universitat Politècnica de Catalunya

Thesis submitted for the qualification of
Doctor from the Universitat Politècnica de Catalunya

Advanced analytics through fpga based query processing and deep reinforcement learning

Gorker Alp Malazgirt

Advisor

Dr. Adrian Cristal Kestelman

November 2018

Vita brevis,
ars longa,
occasio praeceps,
experimentum periculosum,
iudicium difficile.

Abstract

Today, advanced analytics is considered to be an important driving force for evolving businesses and societies. Two fields constitute the foundations of advanced analytics, namely databases and predictive analytics using machine learning (ML). Vast streams of structured and unstructured data have been incorporated in databases, and analytical processes are applied to discover patterns, correlations, trends and other useful relationships that help to take part in a broad range of decision-making processes. The amount of generated data has grown very large over the years, and conventional database processing methods from previous generations have not been sufficient to provide satisfactory results regarding analytics performance and prediction accuracy metrics. Thus, new methods are needed in a wide array of fields from computer architectures, storage systems, network design to statistics and physics.

The aforementioned phenomenon has had an epochal influence on the research of predictive analytics using machine learning. Deep learning based machine learning methods have started to surpass rule-based predictive methods by extracting patterns from vast and diverse amounts of data. Unfortunately, there is no such thing as free lunch, deep learning methods require immense amounts of training and learning. Currently, the computational demands of deep learning methods have been one of the most studied methods in all areas of computing sciences from hardware to all the way up to algorithm design.

This thesis proposes two methods to address the current challenges and meet the future demands of advanced analytics. First, we present AxleDB, a Field Programmable Gate Array (FPGA)-based query processing system which constitutes the frontend of an advanced analytics system. AxleDB melds highly-efficient accelerators with memory, storage and provides a unified programmable environment. AxleDB is capable of offloading complex Structured Query Language (SQL) queries from host CPU. AxleDB is designed to be programmable with a set of special instructions, which enable data movement through memory and storage units and can be programmed from the host CPU. The experiments have shown that running a set of TPC-H

queries, AxleDB can perform full queries between 1.8x and 34.2x faster and 2.8x to 62.1x more energy efficient compared to the state-of-the-art Database Management Systems (DBMSs) such as MonetDB, and PostgreSQL on a single workstation node.

Second, we introduce TauRieL, a novel deep reinforcement learning (DRL) based method for combinatorial problems. The design idea behind combining DRL and combinatorial problems is to apply the prediction capabilities of deep reinforcement learning and to use the universality of combinatorial optimization problems to explore general purpose predictive methods. Most of the engineering, data analysis, and business decision problems can be formulated as combinatorial problems where the optimization process can be described as extracting various model parameters from data or distributions and minimizing errors based on an objective. Thus, TauRieL constitutes the backend of an advanced analytics system.

TauRieL utilizes an actor-critic inspired DRL architecture that adopts ordinary feedforward nets to generate a policy update vector v. Then, the update vector improves the state transition matrix that generates the search policy. Furthermore, TauRieL performs online training which unifies training and searches whereas the current state-of-the-art requires substantial training duration and datasets that precedes the search step. The experiments show that TauRieL can generate solutions two orders of magnitude faster and performs within 3% of accuracy compared to the state-of-the-art DRL on the traveling salesman problem (TSP) while searching for the shortest tour. Also, we present that TauRieL can be adapted to the Knapsack combinatorial problem. With a very minimal problem specific modification, TauRieL can outperform a Knapsack specific greedy heuristics.

Acknowledgements

I am greatly indebted to my advisors Dr. Adrian Cristal and Dr. Osman Unsal for letting me research under their supervision. This thesis would not have been possible without their mentorship, superlative technical guidance and generosity which helped make my time enjoyable while engaging with my research.

I would like to express my deepest thanks and sincere appreciation to my thesis examination panel members and external reviewers for their creative and comprehensive advice until the last draft of my thesis work.

I am grateful to all my colleagues and co-authors at Barcelona Supercomputing Center. Our stimulating discussions, lunchbreaks, sleepless nights and many extracurricular activities will stay as memories to remember for a lifetime.

Finally, I would like to express my sincere gratitude to my family and my grandparents for always showing their love, and imparting to me the importance of education in life. Also, a thousand thanks to all my friends, with whom I have shared moments of joy and sorrow. Their presence has always been significant to me throughout the journey which otherwise might have felt solitary.

This thesis has been supported by Severo Ochoa mobility grant at The Barcelona Supercomputing Center-Centro Nacional de Supercomputación (BSC-CNS), which was accredited as Severo Ochoa Center of Excellence (SEV-2011-0067).

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

ϵ	Update vector rate
γ	Discount factor
κ	An item in the Knapsack Problem
μ	The vector that holds the locations of the centroids
$\nabla_{\theta_{act}}$	The gradient of the expected tour length
ϕ	Permutation, a tour in a given graph
π	The policy that keeps the decisions between state transitions
θ_{act}	Parameters of the actor network
θ_{cri}	Parameters of the critic network
υ	The given value of an item in the Knapsack Problem
A	A set of actions $\{a_1, a_2 \dots\}$
b	The baseline function with respect to the policy
Cl	Cluster representation in the input vector
D	The k-means objective that minimizes distance between the centroids and the elements

f	The function that selects the most likely next state in the transition matrix
G	A given graph that consists of cities
H	The error between the estimations and the baseline
I	Input vector representation to the neural networks
K	The number of episodes that determines the update
L	Total reward
P	Transition probability matrix that determines between two states
p	Probability of a given tour
R	Expected immediate reward after transitioning from a state to another
S	State space in the MDP state space
T	Sample steps
v	The state transition matrix update vector
x	A city in a given graph G
AAU	AxleDB Accelerators Unit
ASIC	Application Specific Integrated Circuit
BSC	Barcelona Supercomputing Center
CNS	Centro Nacional de Supercomputación
DAT	Data Address Table
DBAA	Direct Bus of AxleDB Accelerators
DBMS	Database Management System
DMI	Data Movement Instructions

DNC Differentiable Neural Computer

DPC Data Process Controller

DRL Deep Reinforcement Learning

FDW Foreign Data Wrapper

FIFO First-In-First-Out

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HLS High Level Synthesis

IC Instruction Cache

IP Xilinx Intellectual Property Core

LSTM Long Short Term Memory

MB Megabyte

MDP Markov Decision Process

MIG Memory Interface Generator

ML Machine Learning

ORC Optimized Row Columnar

OS Operating System

PDCS Programmable Data Connection Switch

PGSQL PostgreSQL

PIU Programmable Interconnection Unit

QEI Query Execution Instructions

r Reward which is the negative distance between two cities

RAPL Running Average Power Limit

RL Reinforcement Learning

RTL Register Transfer Level

SATA Serial AT Attachment

SQL Structured Query Language

TSP Traveling Salesman Problem

Chapter 1

Introduction

Advanced analytics is considered to be the most important driving force for evolving businesses and societies. Vast streams of structured and unstructured data have been incorporated, and analytical processes are applied to discover patterns, correlations, trends and other useful relationships that help to partially or fully take part in a wide range of decision-making processes [82]. The data being generated has grown at a rapid pace over the years, and conventional database processing methods from previous generations have not been sufficient to provide satisfactory results regarding performance and prediction accuracy [77, 58].

Consequently, the big data explosion has yielded deep learning based machine learning methods to excel. The large-scale training and approximation from the vast amount of data have surpassed rule-based learning systems [60]. However, the successful solutions require long training times and significant computational demands [53]. This thesis addresses some of the current challenges in the scope above and proposes two methods that aim to meet demanding requirements of predictive data analytics. The objectives, contributions and the details of the proposed solutions are presented, and the insights are shared alongside the experimental results and the conclusions.

1.1 Two components of advanced analytics systems

There are two fields that constitute the foundations of advanced analytics, namely databases and predictive analytics using Machine Learning (ML). The databases have been evolving in order to support a vast amount of high variety data. Furthermore, emerging applications such as autonomous driving and large-scale scientific experiments have created data at an ever increasing rate [82, 39]. The data is extracted and formatted by database queries and queries are processed by query processing units which present the requested data [119]. This ever-increasing rate of raw data creation and the need for intelligent and timely analysis of this data is a challenge. In this endeavor, databases are considered as the frontend entities of the data analytics tasks that are responsible for storing, preparing and preprocessing the data in a structured way.

Simultaneously, predictive analytics apply ML methods and handle the heavy-duty work of generating insights through inference by rigorously processing the available data. Thus they are classified as the backend entities of the analytics systems [103]. Both the frontend and the backend data analytics domains are heterogeneous and complex structures that are composed of different sub-fields. Thus, in the following two sections, we present the challenges facing the front and backend domains and the approaches towards addressing some of the challenges.

1.1.1 The frontend: Query processing

Databases - query processing systems - are capable of storing and querying large amounts of data in various settings from data-centers to autonomous cars [75]. Although there have been alternative general purpose frameworks to databases that do not require explicit data definitions and that hide functions such as data cloning for scalability from the developer [135, 147], these frameworks have not been as successful as replacing databases. The main reason for system architects to prefer databases is due to their ACID

(Atomicity, Consistency, Isolation, Durability) guarantees.

About three exabytes of structured and unstructured data is created and stored in databases each day, and this number is doubling approximately every forty months [81]. Querying enormous amount of data has been a challenge, and new methods have been actively researched and developed for maximizing query processing throughput [5]. Until recently, the majority of query processing research has been targeting software stack on general purpose computing platforms [48] which have also been supported by the Moore's Law that provided performance improvements without major modifications in software [37]. However, the straightforward increase in clock frequencies that led to performance gains has already diminished. Furthermore, powerful uniprocessor general purpose processing platforms have been replaced with multiprocessing architectures. Therefore, homogeneous multiprocessing platforms have been the main choice of computing in database systems in the last decade [37, 56].

Recent advances in specialized hardware accelerators and heterogeneous processing such as Graphics Processing Units (GPUs) [143], Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) have provided significant improvements in computational power [97]. Also, the massive increase in data sizes, stringent design requirements and the mentioned paradigm shift in the computing domain towards specialized and heterogeneous processing are causing query processing systems to reinvent themselves by adopting a unified domain-specific hardware/software approach [25].

Classical control-flow-based query processing engines deliver lower computational throughput compared to what can be achieved by application-specific hardware. From one side, to alleviate the overheads of data movement, one promising solution is to bring the computation closer to where the data resides, so that more operations can be completed avoiding non-essential data movement [124]. In this method, the gains are two-fold: easing the load on the host CPU for performing database operations, and reducing the negative impact on the performance of high-latency I/O operations. As a result, significant throughput improvements, as well as reduction of I/O

overheads can be achieved [108]. On the other hand, streaming data through highly specialized hardware accelerators in a deeply pipelined fashion can significantly improve the computational throughput of the query processing engine.

FPGAs provide a unique opportunity to build an efficient query processing platform, by constructing a high-throughput execution engine with the additional aim of minimizing overheads of data movement. It is mainly the consequence of; (i) the inherent characteristics of massively parallel and configurable architecture of FPGAs, suitable for data streaming in deep pipelined-style execution (ii) the rise of High-Level Synthesis (HLS) technology, which makes FPGA applications relatively more straightforward to develop compared to low-level languages such as VHDL or Verilog, and (iii) the availability of soft cores that allow rapid generation of interface protocols such as PCIe 3.0 (Peripheral Component Interconnect Express) or SATA-3 (Serial AT Attachment) on FPGAs.

In a nutshell and within the context of the frontend, the objectives of this thesis can be listed as:

- To design a set of energy efficient and high-performance query processing hardware accelerators for FPGAs using high-level synthesis methods for rapid prototyping

- To provide an infrastructure in FPGA that sits between a host CPU and SSD storage, thus bringing computation closer to data and enabling query processing hardware accelerators for efficient, high-performance query processing in a single compute node

Based on the objectives, we present the thesis contributions in Section 1.2.

1.1.2 The backend: a predictive method by combining deep reinforcement learning and combinatorial optimization

The second field that constitutes the foundations of analytics is the predictive analytics using deep learning methods. The predictive analytics enable to extract patterns from datasets and aim to provide the best assessment of the likelihood of events whether it be predicting a pedestrian in shortest path in map, grouping photos in folders, or forecasting financial markets. Deep learning architectures are composed of neural networks, densely connected processing elements that are loosely named after the primary information-carrying cells of the brain, the neuron [71].

Deep learning is separated from the rule-based ML systems such that patterns in the data are not extracted based on a set of features [21]. Moreover, deep learning architectures do not necessitate handcrafted feature extraction that in most cases require domain knowledge. Instead, an end-to-end training process which is called supervised learning which exploits the vast amount of labeled input-output data. Using supervised learning deep learning models can extract features and apply transformations to map inputs to outputs [42]. Thus, deep learning frameworks are built from sampling the inputs and probabilistic distributions with expected outputs. However, the end-to-end training procedure requires large amounts of input data for successful outcomes [42].

The trade-off of not requiring handcrafted features while requiring more input data has turned out to be favorable for deep learning architectures [71, 86, 129]. Deep learning algorithms have been shown to outperform heuristics in various application domains. However, the vital necessity for high volumes of data for successful predictions also hinders broad adoption of deep learning in various domains [122, 19, 116]. There does not always exist an adequate amount of data for successful training or high costs, noisy processes and computational limitations hinder the generation of datasets [28, 105].

A critical domain that is affected by the mentioned phenomenon is the combinatorial optimization problems [7]. These problems are computationally challenging either to search for the optimal solutions due to their exponential exploration space or to generate a single feasible solution. Also, the accuracy of predicting combinatorial optimization problems are also affected when available datasets possess a large amount of sub-optimal instances. In this case, although the training is performed successfully, the prediction performance may be unsatisfactory. Nonetheless, there exists a branch in the machine learning field that does not rely on supervised learning methods, namely reinforcement learning (RL).

Deep Reinforcement Learning (DRL) combines deep learning with RL [11]. In essence, RL is a technique that consists of an agent, which is a software program that learns from its environment by taking actions (decisions), collecting rewards and adjusting its behavior based on the rewards. RL utilizes dynamic programming and Markov Decision Process (MDP) methods which allow the agent to explore the environment through taking various actions (exploration), and take advantage of recorded experiences by replaying previous actions (exploitation), by and large stochastically and in a sequential manner [121, 91]. DRL architectures have incorporated complex pattern recognition and approximation capabilities of deep learning for representing agent policies and predicting value functions [86]. The significant difference of RL from supervised learning based deep learning is that RL systems learn from experience by sampling from the state space whereas supervised learning occurs through approximations over labeled datasets [63]. Hence, by representing agents' possible actions as deep neural networks, the whole reinforcement learning can be trained end-to-end that allows DRL architectures to benefit from exploring the available state space and collect experiences whereas the success of deep learning architectures only relies on the given datasets. Furthermore, sequentially exploring the state space yields DRL architectures to target problems that advance sequentially in order to reach a solution, such as planning problems [123]. Henceforth, combinatorial optimization and natural language processing problems can be listed among these types of problems [22, 24]. Although, deep reinforcement learning architectures can

modify internal states and consider output feedbacks, the overall training times of deep reinforcement learning architectures have been as high as deep learning architectures [112].

The main of the advantages of DRL in the scope of this thesis that they are capable of tackling combinatorial optimization problems, which form the basis of the most prediction problems in engineering, data analysis, and business [94]. A solution can be deduced by combining sample space exploration and available nonoptimal datasets. The rewards collected during explorations are then used to extract the parameters by minimizing predetermined error measures concerning an objective function [110, 36]. Hence, the universality of combinatorial optimization problems combined with the strengths of deep reinforcement learning is a promising combination and constitutes our design idea towards designing the predictive analytics backend. Thus in this thesis, the objective towards building a general backend for predictive analytics can be itemized as:

- To develop a deep reinforcement learning method and tackle combinatorial optimization problems
- To apply the proposed method to two different combinatorial optimization problems for displaying generality of the solution
- To eliminate the main weaknesses of deep learning which are the long training durations and the sheer necessity of labeled datasets

Next section presents the contributions of the thesis based on the objectives that we have described in Sections 1.1.1 and 1.1.2.

1.2 Thesis Contributions

The previous chapter introduced the emergence of advanced analytics and their effects on the computing and machine learning landscape. In this thesis, we aim to address the challenges mentioned above and present our contributions. For the analytics frontend, we present the design and the implementation of AxleDB, a novel architecture for managing data movement through hardware accelerators and storage. AxleDB is an FPGA-based query processing system that melds highly-efficient accelerators with memory, storage and provides a unified programmable environment. AxleDB is designed to execute complex Structured Query Language (SQL) queries in full by performing various time-consuming query operations using FPGA based hardware accelerators, while the data movement between storage and compute units are handled with custom move instructions. Designing AxleDB presents the following contributions:

- FPGA based query processing accelerators are presented for filtering, arithmetic and aggregation operations which are designed to speed up database analytics for in-memory databases. Unlike traditional FPGA design methods, our hardware accelerators are composed using High-Level Synthesis (HLS), which enables high-level descriptions of query processing functions to be targeted directly into Register Transfer Level (RTL).

- A unified AxleDB platform is presented that includes query processing-specific accelerators and the efficient data management mechanism to control the flow of data, which effectively enables rapid query processing in the hardware.

The performance and energy efficiency of the AxleDB is tested under various conditions, by running five decision-support TPC-H queries. We compare AxleDB to state-of-the-art software-based DBMS, PostgreSQL, and MonetDB, in the single-threaded and multi-threaded modes in a single compute node. Scaling data for multiple compute nodes has not been investigated in this thesis.

Moreover, for the analytics backend, we propose TauRieL [1], a DRL based method for solving combinatorial optimization problems, and our contributions can be listed as the following:

- TauRieL searches for optimal solutions given a combinatorial optimization problem by training ordinary deep neural networks in a reinforcement learning setting and can generate solutions two orders of magnitude faster than the state-of-the-art while maintaining similar prediction accuracy.
- TauRieL introduces an explicit state transition matrix that complements the learning process by generating samples solutions from the transition probabilities.
- A clustering and merging scheme is developed in order to divide the given problem into subproblems for coping with the large design space.

The contributions of TauRieL is shown by solving two innately different combinatorial problems, namely the traveling salesman problem (TSP) and the Knapsack problem [94]. We compare the prediction accuracy, training and inference times when the problems start from scratch.

[1] A wood-elf character from Hobbit the Movie who possesses *superior senses and pathfinding skills compared to humans*

1.3 Organization of the thesis

In this chapter, we have discussed the emergence of advanced analytics and the future challenges in the domain. Then, we have introduced the two entities, namely databases and predictive analytics using deep learning. Also, we classify and identify their roles as the front and backend of advanced analytics. Before explaining the objectives of the front and backend, we have introduced the motivations and the challenges.

After listing the objectives for each part, we have presented the contributions in Section 1.2. In the next chapter, we scrutinize and present the background works. Chapters 3 presents the details and the experimental results of AxleDB. Chapter 4 starts by reiterating the problem at hand and introduces TauRieL. The background, the search algorithm, and the preliminary results are shown on TSP. The future work which we discuss in Chapter 5 will enlarge Chapter 4 by introducing the Knapsack problem. Chapter 6 finalizes the thesis by presenting conclusions.

Chapter 2

Background

In this Chapter, we discuss the relevant state-of-the-art of data analytics front end (Query Processing, Section 2.1) and backend (Predictive Analytics, Section 2.2). We also comment on the thesis advances beyond this state-of-the-art, wherever appropriate.

2.1 Query processing systems

2.1.1 Query processing hardware design

There have been two main efforts to implement query processing on hardware, namely ASIC-based and FPGA-based solutions [141, 62]. The work in [141] presents an ASIC-based method with heterogeneous compute tiles which manipulate rows and handle query processing operations in a coarse grain way. In order to manipulate streams of data according to the given query, the authors present spatial and temporal planning which enables/disables compute units along a predefined data path.

The most significant difference between full custom ASIC design compared to HLS-based FPGA designs is the flexibility. HLS is the process of generating low-level cycle accurate RTL specifications from high-level structures which are generally in C/C++. Different sizes of memory or compute structures from database columns can be optimized by extending or shrinking data sizes in the high-level source code. Next, these structures are programmed

onto the logic and memory blocks of the FPGAs [30]. Consequently, HLS-based FPGA designs are a more flexible and productive solution for rapid design. However, ASIC-based solutions excel in performance compared to FPGA-based solutions because FPGA hardware provides programmable logic and memory blocks, and these blocks must be programmed efficiently for the required functionality. On the other hand, ASICs can be designed from scratch, maximally tailored for the application at hand [115].

While the HLS methods propose hardware generation from high-level programming languages such as C/C++, Glacier [88] compiles VHDL code from algebraic expressions which adds additional steps in the system design because algebraic expressions must be created from SQL expressions or they are taken directly from a query planner. Our proposed method in this thesis binds accelerators to SQL operators semantically. Then, the query plan is made accordingly.

Accelerator affinity with its host is one of the essential components in the hardware accelerator domain. Loosely coupled accelerators can execute independently from their host processors, and this loose coupling allows them to service to different processors. On the other hand, highly coupled accelerators are attached to a particular system and cannot be used by any other service. The authors of [95] discuss efficient methodologies for decoupling query processing hardware from a general purpose host computer this in contrast to in-memory database acceleration where the memory of the accelerators is controlled by a host system as in AxleDB developed in this thesis [111].

Runtime reconfiguration capabilities of FPGAs have allowed runtime query processing customization. Authors of [34] have built a database operations library which at runtime forms the data path based on the given SQL query. They have focused on data filtering operations. The main advantage of runtime reconfiguration is to eliminate the synthesis of queries if the available runtime operator library can execute the given query. The flexibility that HLS provides is at compile time rather than runtime. Hence, there is a possibility to combine runtime reconfiguration with HLS technology such as the newest Xilinx SDAccel system design tool [2]. Although FPGA based processing systems

could be enhanced by the use of dynamic reconfiguration, the reconfiguration requires additional glue logic on FPGA and a predetermined selection of the functionality that will be modified during the runtime. AxleDB does not employ runtime reconfiguration. Instead, it is designed to be programmable via a set of instructions for handling different database queries.

2.1.2 Designing query processing engines

Previous studies have looked into designing efficient query processing engines, employing vector architectures [49], ASICs [142], GPUs [104] or CPUGPU-FPGA [50, 9] proposals. On the other hand, other approaches either used FPGAs statically [27, 96], or leveraged dynamic reconfiguration to better fit the requirements of each query [16, 66, 149]. We differ from these works by following a static but programmable approach in query processing and data management, as we can support as many operations as we need, without requiring runtime reconfiguration. The industry has also invested in a few products, IBM Netezza [59] and XStream Data dbX [113], which offer full DBMS solutions.

We compared AxleDB with a set of state-of-the-art FPGA/ASIC-oriented query processing platforms: Ibex [139], Q100 [142], BlueDBM [61] Sukhwani et al. [119] and Jaeyoung et al. [35]. Ibex [139] is a database storage engine that is equipped with a limited set of query processing operations, working directly with data inside SSD. Q100 [142] proposes domain-specific database processors, but without supporting data management from off-chip storage. BlueDBM [61] proposes a system architecture with flash-based storage and in-store processing capabilities, but it is not specialized for query processing. Sukhwani et al. [119] present an FPGA-based query processing engine that is attached to a DBMS via PCIe-3 with a data compression capability. Jaeyoung et al. [35] present a smart SSD that incorporates it with memory and low power embedded processing units inside the SSD controller. Although the computation is closer to the storage, the compute blocks are not specialized for query processing and necessitate a more constrained programming style for the processors such as for prevention of register spilling. The work in

Table 2.1: Comparing AxleDB with state of the art platforms, in terms of accelerators

	AxleDB	Ibex	Q100	BlueDBM	Sukhwani et al.	Jaeyoung et al.	Ziener et al.
filter	✓	✓	✓	×	✓	✓	✓
aggregation	✓	✓	✓	×	✓	✓	✓
hash join	✓	✓	×	✓	✓	×	✓
merge join	×	×	✓	×	×	×	✓
order by	✓	×	✓	×	✓	×	✓
DB indexing	✓	×	×	×	×	×	×
compression	×	×	×	×	✓	×	×

[119] presents a query processing system that efficiently uses partial dynamic reconfiguration capabilities for on-the-fly query processing.

Table 2.1 lists the embedded accelerators for each of the studied platforms. AxleDB currently covers most of the necessary modules to run complex queries, although operations such as pattern matching or compression are not supported yet. On the other hand, as illustrated in Section 4, we proposed a novel and efficient accelerators for many important SQL query primitives using modern HLS tools. Although the hash join, sorting and indexing engines that are used in AxleDB are out of the scope of the thesis, we present and compare all the features with state of the art in 2.1. The detailed explanation of the blocks above can be found in [111].

Ibex: AxleDB differs from Ibex in two ways. First, Ibex does not provide any programmable data movement support. Also, the set of hardware accelerators are limited. AxleDB both provides data move instructions and also provides aggregation and sorting. In complex queries, ibex fallbacks to the host CPU. Thus, there may be scenarios where the fallbacks can be separated in query execution and creating a lot of data movement between the accelerators and the CPU. AxleDB supports software fallbacks when as the last step in the query schedule. Hence, if given SQL query provides it, AxleDB sends operands to the host CPU which results in the final product after the last operation is executed in the host CPU.

Q100: AxleDB differs from Q100 in micro-architecture and ISA design. Instructions of AxleDB are centered around data movement and initiating

the accelerators, whereas Q100 has SQL-style instructions. The authors based their micro-architecture design on the sensitivity analysis of TPC-H queries. However, the off-chip bandwidth experiments have shown that query execution speeds are affected to a greater extent. In contrast, our platform is designed to maximize the available off-chip bandwidth by explicit data movement instructions. Thus, this allows finer grain data movements for execution. Also, as Q100 is tailored as a composition of a particular purpose (ASIC) blocks, thus, extending the blocks can be an expensive process.

BlueDBM: Its single node consists of flash storage, accelerator hardware in FPGA, flash controller and network interface. Data requests are sent from the host with minimum kernel overhead. Specialized accelerators process the data retrieved directly from flash storage, bypassing DRAM. AxleDB also supports different types of data movement, allows data streaming among the host, SSD, DDR-3, and accelerators. AxleDB is specialized for database query processing using an efficient set of query processing accelerators, whereas BlueDBM does not support the performing of such complex SQL queries. BlueDBM has a distributed structure that allows them to scale up more processing nodes, providing a larger address space. This capability is not yet supported in AxleDB, through multiple nodes of AxleDB platform can be enabled, this procedure requires additional modifications such as data partitioning and query scheduling [44].

Sukhwani et al.: [119] present an FPGA-based query processing engine. The engine is attached to a DBMS via PCIe-3. The supported functionality of the accelerator is filtering, join and sorting. To improve the throughput, the data is compressed by the host. Therefore, the decompression is the first step in the query processing pipeline on FPGA and processed queries are sent back in decompressed form. The authors do not provide any indexing mechanisms, and all queries that are sent to the accelerator require a full table scan. Also, the join and sorting units are not streaming based. Thus they use onboard DRAM for storing intermediate results. In the query pipeline, the filtering units always come before the join/sort units. Necessary data is read from the DRAM of the host. Thus, this requires additional data movement from external storage to DRAM by either the host or the accelerator. In AxleDB,

the data movement and acceleration instructions allow processing blocks to execute in any order. Hence, it is possible to utilize a join/sort unit before filtering. AxleDB is designed to interface external memory directly and stores intermediate data in FPGA's memory.

Jaeyoung et al.: [35] present a smart SSD that incorporates SSD storage with memory and computing resources inside the SSD controller. This design allows internal aggregate I/O bandwidth to be 5X higher than fastest SAS and SATA architectures. The Smart SSD architecture presented consists of embedded processors that are on the SSD host interface controller. They are coupled with DRAM and SRAM memories for intermediate data storage. NAND memory arrays inside SSDs allow parallel access. Contrary to Smart SSD, AxleDB is designed to work on an FPGA, and it is connected to an SSD via SATA port. This allows the design of specialized accelerators on the FPGA rather than general purpose embedded processors. In two cases, smart SSDs can fail to exploit the advantages of its architecture, because these scenarios might not require extensive communication between the embedded processors and the SSD units. The first case is when the embedded processors require data communication between the host, and the utilization of the SSDs are very low. Next, the general purpose local memories of the embedded processors do not satisfy the memory requirements for the problem at hand and it causes register spilling and decrease the performance. In these two cases, specialized accelerators can provide better results compared to general purpose embedded processors of the smart SSD.

Ziener et al.: [16] present a query processing platform that leverages the partial dynamic reconfiguration capability of FPGAs to fit the requirements of each query better on-the-fly. Query primitives such as filtering, aggregation, hash join, and sorter are gathered in a library while supporting both column- and row-oriented data storage formats. However, this work does not follow a direct-SSD-coupled approach that diminishes the overall throughput, which can suffer from data offloading and partial reconfiguration overheads.

2.2 Predictive analytics

In this section, we aim to present the relevant information about predictive analytics and correspondingly allow readers to get acquainted with the foundations which can be categorized into three. First, in the next section, we first describe deep neural networks and reinforcement learning which enable prediction through exploration of the state space and learning from available datasets. Second, Section 2.2.2 presents the combinatorial optimization problems which lie in the heart of many engineering and scientific settings. Thus, we scrutinize previous works that aspired to produce optimal results tailored for particular combinatorial optimization problems. Finally, we target TSP which is a well known combinatorial optimization problem, and it has been discussed to be a hard problem for classical and quantum computing [134]. Hence, in Section 2.2.3 we introduce previously published that target TSP. Under those circumstances, a successful TSP solver could provide a step closer to a general predictive analytics system by transforming the given problem to TSP or a variant of it. However, the transforming process is not in the scope of the thesis, the problem reduction techniques have been well studied [70]. Therefore, in Section 2.2.4, we explain the existing deep learning based efforts that target TSP and compare and contrast with our proposed method in this thesis.

2.2.1 Background in deep neural networks

Deep learning allows to extract complex patterns and function approximations and to map an input to output without any pre-processing, namely the enigmatic feature extraction processes [57]. The training process is called supervised learning, and the backpropagation is the underlying method. Backpropagation algorithm in an optimization setting minimizes generic loss metrics (e.g., squared loss, cross entropy) by modifying the gradients [72]. Convolutional Neural Networks and Recurrent Neural Networks are one of the most successful architectures that have provided breakthrough results in image processing, natural language processing, etc. through supervised learning [148, 76]. Furthermore, the design process of these architectures

concentrates on amending neural networks as opposed to tailoring heuristics for application specific purposes. Also, previous works have shown that a well tuned neural network can be reused liberally as high-level feature extractors [52].

Recently, Differentiable Neural Computer (DNC), a deep learning architecture which differs from previous networks has been proposed [45]. The main characteristic of this model [45] is that the authors provide an external memory matrix complementing the Long Short Term Memory (LSTM) cell state [55]. Thus, the external memory matrix is trained to support the existing recurrent neural network. The authors show that the hybrid model can be trained as a regular neural network including the external memory matrix. The examples demonstrate solutions to problems such as finding the shortest path and solving moving blocks puzzle. Although the proposed architecture possesses a built-in memory structure that resembles a random access memory, the proposed solution can still be classified as a supervised learning architecture.

Unlike a general purpose computer, the most suitable environment for DNC [45] to excel is when the amount of data to predict, classify or infer an algorithm is high. Also, the DNC solution suffers from high training duration and performs poorly when the final algorithm to reach unclear, or the result requires interactions with the environment [121]. For situations that do not favor the DNC, DRL architectures that combine reinforcement learning and neural networks, are shown to perform well [85, 86, 129].

Reinforcement learning (RL) is a method that consists of an agent, which is a software program that learns from its environment by taking actions (decisions) and adjusting its behavior based on rewards. RL representations have utilized dynamic programming and Markov Decision Processes (MDP) which model the agent and its interactions with the environment through describing actions and states. In almost all cases, there is an agent that takes different actions (exploration), and take advantage of recorded experiences by replaying previous actions (exploitation). Furthermore, the actions by large are stochastic, partially observable and happen in a sequential manner [121, 91].

DRL architectures have incorporated complex pattern recognition and approximation capabilities for the agents and the rewards from the environment by employing neural networks [86]. Hence, agents make decisions either through maximizing future rewards from experience or by extracting policy mechanisms based on state action patterns [85, 90]. Thus, the significant difference of RL from supervised learning is that RL systems learn from experience whereas supervised learning occurs through approximations over labeled datasets [63, 71].

2.2.2 Previous work for solving combinatorial optimization problems

Combinatorial optimization is a branch of mathematical optimization, and it has been a cornerstone research field with various application domains from biotech, finance to manufacturing. Many problems are arising in this field that does not yield optimal solutions with polynomial-time algorithms, and this constitutes the main reason for continuing interests and contributions. A set of reducible problems described by Karp et al. [64] laid the foundations of combinatorial problems, and subsequent research on dealing with computationally intractable algorithms through approximation methods with empirical performance estimates [40].

Designing approximate methods for combinatorial optimization problems can be classified into three. These are exact solutions, approximations of exact solutions and the design of heuristics. The exact solutions have been proposed through designing mathematical programming methods such as Simplex Method [12]. Generally, exact solution methods have scaled poorly due to computational costs. Thus, approximations of exact solutions have been proposed [12]; Lagrangian relaxation and approximations on the upper bound and the optimality gap are among the best-known methods [20].

Third and the most popular method has been designing heuristics which are designed to generate useful solutions without any guarantees for the optimum. Traditional heuristic design can be divided into three subclasses. The first class of methods is called the constructive method where the heuristics are

constructed from null and finalize when a feasible solution is found, such as the nearest neighbor heuristic [32, 107]. The second method is called the improvement heuristics where the heuristic tries to develop better results by improving current solution based on rules. N-opt heuristic for TSP problem can be given as an example [51, 15]. The third method is called a hybrid which unites the previous two methods. Heuristics tend to exploit local improvements with constructive searches [14, 109].

Though heuristic approaches have been shown to work well, each approach requires domain knowledge and application specific tailoring. Specifically, each heuristic requires tailored configurations which are in essence the problem representation and the playout operations that result in the next step of the heuristic. These configurations vary widely from choosing suitable abstract data types and containers to computational costs of next-state playout.

Throughout the years, metaheuristics have emerged as a class where they allowed a mixture of stochasticity and exact solution proposals that can be applied directly to sample space. Furthermore, more generic algorithmic recipes allowed metaheuristics to span a range of optimization problems. Evolutionary computation, simulated annealing, and tabu search are known examples in this area [41].

Hybrid metaheuristics have also been researched for compensating for weaknesses of the constructive improvement heuristics. Authors in [74] propose hybrid simulated annealing and tabu search metaheuristics for solving TSP. They explored local neighboorhood search for sub tours with tabu search whereas global search has been complemented using temperature-based simulated annealing. Towards providing more generality, four parameters are defined that are proposed as the only parameters for customization for different problems. Thus, application heuristics parameters such as epoch length, candidate solutions and cool time are defined regarding global variables such as the number of cities, standard deviation.

Another hybrid method [118] combines construction heuristics such as nearest neighbor search and tour improvement such as Lin–Kernighan [ref] local search with evolutionary computing for TSP [133]. Crossover operations which are the essential operations in evolutionary computation use an edge

based crossover operation where similar tour paths are promoted to be explored in the gene pool [92]. The authors have shown suboptimal results for the larger input sets of the TSPLIB library.

2.2.3 The impacts of the traveling salesman problem, a well known combinatorial optimization problem

Solving combinatorial optimization problems has an immense impact in all fields. For example, TSP which is a well known combinatorial optimization problem has been developed for solving a wide array of practical and theoretical problems. Examples include the use of TSP in music for conjunct melody generation in computer-aided composition, as well as for forming automatic playlists and track/artist suggestions - now used by Spotify and Tidal [102]. TSP heuristics have also been used for diffractometer guidance in X-ray crystallography [22]; and for Telescope scheduling in exoplanet discovery [67] as well as in galaxy imaging [26]. It can also be applied to bioinformatics as a genome ordering problem by posing the ordering as a path traveling through each gene marker [4] or as a clustering problem to solve gene expression clustering [31]. Last but not least, it can be leveraged for its original application of finding the shortest tour in a classic map, for example as a Traveling Good Samaritan Problem for the Meals for Wheels charity program in Atlanta [15].

2.2.4 Targeting TSP using neural networks

Previously, TSP algorithms have employed neural networks in order to complement local search heuristics [114, 132]. Authors in [114] use Kohonen networks and devise a TSP solver. Self-organizing nature of Kohonen networks iteratively executes and tries to map neurons which are dispersed onto the 2-D Euclidean plane to the cities. At each iteration, neurons attempt to decrease the distance between the cities in the neighborhood region which is a parametric set that consists of the nearby cities. The learning rate and the neighborhood function are the hyperparameters. Because the number of neurons is higher than the cities, the algorithm presents an ordering mechanism

for selecting the best fit neurons that represent the cities.

The requirements of the vast amount of customization in optimization problems and recent advancements in neural network architectures have caught the attention of machine learning community. Vinyals et al. have proposed pointer networks which are attention based sequence-to-sequence learning architectures and presented results on TSP [131]. Without significant hyperparameter explorations, proposed architecture has managed to generate competitive results for other problems such as delaunay triangulation. However, the problem sizes have not been as large as state-of-the-art [8] which have employed TSP specific local search moves [118] and provided optimal solutions up to thousands of cities.

Bello et al. [17] recently proposed a TSP solver RL framework based on neural networks. Pointer-networks have been employed for policy gradient and expected tour length prediction. The sequential nature of the framework resembles tour construction algorithms. Stochastic sampling and the actor-critic architecture updates the expected tour length with the current policy (on-policy), and the gradient updates are worked out using reinforce algorithm [137]. Also, a separate neural network for tour exploration incorporates an expected reward based value iteration approximation. All neural networks are constructed using pointer networks [131]. Both pre-training and random initialization of the weights are realized as the initial starting state. The framework presents improved tour lengths and execution times compared to Christofides and supervised learning methods [29, 131].

The authors of [65] present a deep Q-learning [85] and a graph embedding based solution to target combinatorial optimization problems. Given a problem as the graph, they first perform function mappings such as belief propagation that learns feature vectors from latent variable models. Then, the learned embeddings allow learning a construction graph building heuristic. The Q-learning allows constructing the solution based on the reward which is defined as the change in the cost function among the candidates. A helper function is also employed that helps to satisfy the constraints of the combinatorial optimization from the partial solution throughout the construction process.

Table 2.2: Comparing TauRieL with previous deep learning based solutions

	TauRieL	Vinyals [131]	Bello [17]	Khalil [65]	Kool [69]
Feed Forward Nets	✓	×	×	×	✓
Recurrent Nets	×	✓	✓	✓	×
Supervised Learning	×	✓	✓	✓	✓
Reinforcement Learning	✓	×	✓	✓	✓
Policy Based RL	✓	×	✓	×	✓
Value Based RL	×	×	×	✓	×
Input Encoder	×	✓	✓	✓	✓
Output Decoder	×	✓	✓	✓	✓

Kool [69] proposes a TSP solver framework based on the attention model [130]. The attention mechanism consists of the input encoder and output decoder. The input encoder uses multiple attention mechanisms to compute the embeddings of the input nodes. Given an input set, the output decoder calculates a probability distribution over the input nodes using attention. The decoder includes a context embedding mechanism which includes all the node embeddings in addition to the graph embedding is defined as the mean of all the node embeddings that are generated at each hidden attention layer. In order to calculate the output probabilities of the next city in the permutation, the decoder requires the embeddings of the start city as well as the graph embeddings which displays that proposed solution is permutation variant similar to recurrent neural network based solutions. Although the results for 50 and 100 cities have improved compared to Bello et al. around 2% and 1% respectively, the reported training times have been higher than Bello et al. [17].

We compare TauRieL and the aforementioned deep learning based methods in Table 2.2. The row in the table present the features that are specific to each proposed method. For each feature, the differences and similarities of each method can be explained below:

Feed Forward, and Recurrent Nets: Although Kool [69] and TauRieL are the two architectures with feedforward layers, Kool [69] also employs attention networks that are more complicated than the conventional feedforward neural nets. Thus the attention networks are dominant blocks in the presented architecture. However, TauRieL only uses conventional feedforward neural

networks, and its architectural complexity is the lowest among all the proposed methods. The rest of the proposed solutions are based on recurrent neural nets. Vinyals [131] and Bello [17] employ almost identical architectures and Khalil [65] suggests an architecture that resembles unrolled recurrent neural nets. Regarding architecture complexity, the recurrent nets surpass feedforward nets.

Supervised and Reinforcement Learning: Vinyals [131] proposes to train its network with supervised learning on various sizes of TSP problems, and Bello [17] extends this network by first training the network with supervised learning and applying REINFORCE [121] training in a reinforcement learning setting. Khalil [65] employs supervised training for generating graph embedding and then uses reinforcement learning to training the deep Q network. Similar to Bello [17], Kool [69] and TauRieL uses REINFORCE [121] algorithm to train the neural networks.

Policy and Value-Based RLs: All the works that we list in Table 2.2 except Khalil rely on policy based methods where the neural networks represent the policy. Khalil [65] employs neural networks in order to represent and approximate the Q-score [121] for constructing the graph sequentially from the embeddings for the problem at hand such as TSP.

Input and Output Decoders: All the works except TauRieL and Khalil [131] employ input-output decoder neural networks. The works proposed by Vinyals [131], Bello [17] and Kool [69] are inspired by language attention nets and seq-to-seq models [130, 120] which employ recurrent neural networks for encoding the input words sequentially and decoding for inference. Khalil [65] presents an embedding algorithm that is inspired by word2vec [84] architecture and aims to generate a vector space from graphs. TauRieL proposes to represent the latent exploration space by storing state transition probabilities explicitly in a state transition matrix. Thus, it does not make use of input encoders or output decoders.

Chapter 3

AxleDB: A novel programmable query processing platform on FPGA

In this chapter, AxleDB which is one of the main contributions of this thesis is presented. AxleDB is a FPGA query processing platform which encapsulates database-specific hardware accelerators with memory and storage in a unified programmable environment. In the next section, we introduce the overall architecture of the AxleDB. The overall architecture also includes describing AxleDB's major components. In the following section, an example query is presented to show how its components are utilized to process the example query. Section 3 introduces the FPGA based filtering, arithmetic, logic and aggretgation hardware accelerators. Those accelerator units, which are the most important components of AxleDB, are presented in depth. The rest of the accelerators are introduced and briefly described as they are not in the scope of the thesis. The experiments are presented in Section 4. This section begins with evaluating the performance of the query accelarators in scope. The rest of the experimental results include system-level comparisons of AxleDB to the state-of-the-art.

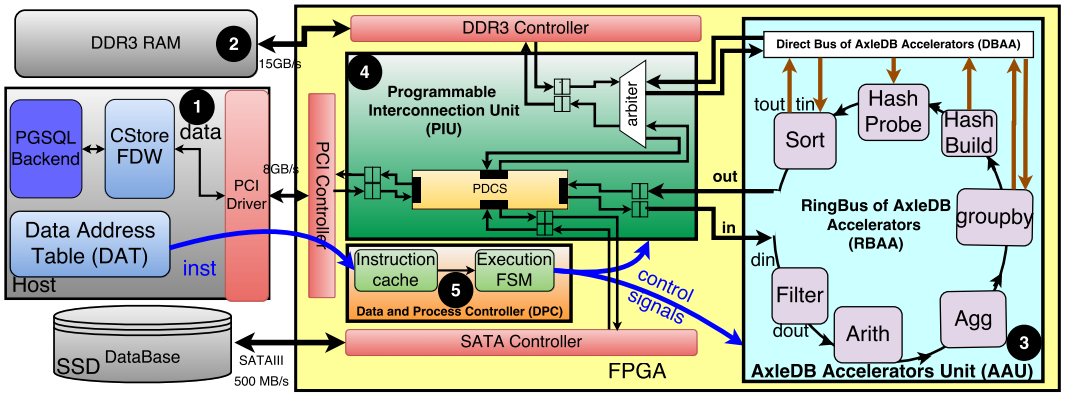


Figure 3.1: Overall architecture of AxleDB with its major components. ❶ software extensions for DBMS in the host, ❷ Data storage units and device controllers, ❸ a set of efficient query processing accelerators, which altogether form AxleDB Accelerators Unit (AAU), ❹ Programmable Interconnection Unit (PIU) to manage the accesses to the off-chip data storage units, in a fully flexible fashion, ❺ Data and Process Controller (DPC) to orchestrate the involved modules of AxleDB to process SQL queries.

3.1 Architecture of AxleDB

The main principle of the proposed database query processing platform, AxleDB, is to essentially move database computations closer to where the data resides, to obtain high performance in a flexible and programmable environment. Figure 3.1 shows the overall architecture of AxleDB. AxleDB resides between the host machine that runs the DBMS, and the database storage in an SSD. In the host, we primarily targeted to use PostgreSQL [98], one of the most popular open source relational DBMS. However, the infrastructure of AxleDB was designed to be software-agnostic and could be ported to other DBMS, e.g., MonetDB [87].

The host communicates with AxleDB through an Application Program Interface (API), to transfer data and instructions using the PCIe-3 interface. When the host initiates the query execution, the query plan needs to be converted into AxleDB instructions. Inside AxleDB, these instructions are managed and executed by the Data and Process Controller (DPC) ❺, which orchestrates the movement of data blocks between SSD, DDR-3, host, and Accelerators. The query is effectively executed by streaming blocks of data,

from the storage, through the accelerators, and back. Finally, the result of the query is returned to the host. The architecture of AxleDB is composed of five major components which are explained thorougly in the coming sections.

1. ***Software Extensions for DBMS in the host*** ❶, including the Data Address Table (DAT) and the CStore Foreign Data Wrapper (FDW) extension of PostgreSQL, to manage the transfer of instructions and data, respectively.

2. ***Data Storage Units*** ❷ , i.e., SSD and DDR-3 are used as the primary or secondary database storage units and device controller cores, i.e., SATA-3, DDR-3, and PCIe-3 to manage the data transfer to/from storage units.

3. ***AxleDB Accelerators Unit (AAU)*** ❸, which is a set of efficient DBMS query accelerators, i.e., filter, arithmetic, aggregation, group by, hash build, hash probe, and sort. To transfer data, accelerators are organized inside a ring bus, the RingBus of AxleDB accelerators (RBAA), and a direct bus, the DirectBus of AxleDB Accelerators (DBAA).

4. ***Programmable Interconnection Unit (PIU)*** ❹, to set up a path to transmit the data in a fully flexible fashion. The PIU is composed of ***i)*** a 4-port bidirectional programmable data connection switch (PDCS) to exchange the data among SSD, DDR-3, host, and RBAA, ***ii)*** an arbiter to control the bandwidth sharing of DDR-3 by serializing its concurrent requests, and ***iii)*** a set of synchronizing First In First Out (FIFO) buffers for each individual port, separately for read and write directions, to cross the different clock domains.

5. ***Data and Process Controller (DPC)*** ❺ ,that is composed of an Instruction Cache (IC) to locate the DBMS specific instructions and an Finite State Machine (FSM) ***i)*** to manage the accesses to the off-chip data sources and ***ii)*** to control the accelerators to execute the corresponding query, by issuing the appropriate control signals to the

PIU and to the AAU, respectively. These signals are generated by translating instructions of AxleDB.

3.1.1 Major Components of AxleDB

In this section, we elaborate the architecture of AxleDB and describe the role of each constituting component, individually.

DBMS Software Extensions for AxleDB

The host communicates with AxleDB for two purposes: ***i)*** to access the database tables that reside in the SSD, and ***ii)*** to program AxleDB using query-specific instructions in order to execute the SQL queries. In AxleDB, to process complex SQL queries, these queries first need to be translated to our specific instructions.[1] However, the certain currently unsupported operations, such as floating point division (DIV), can utilize a fallback-to-host scheme. Currently, AxleDB supports this if the unsupported operation can be scheduled as the last operation which does not require significant effort to reschedule the query. However, data management and query scheduling gets complicated for more complex software fallback scenarios when AxleDB and host CPU need to send and receive data during the fallback operations.

We use the CStore FDW extension of PostgreSQL to access the database tables [33], which we refer to as 'CStore' for short. CStore manages data in a column-oriented format [117] that cause discarding unnecessary loads during the query processing and provides better I/O utilization. To transfer these instructions, we use a shared memory region between the host and FPGA, called Data Address Table (DAT). DAT resides in the host memory and holds the list of instructions of AxleDB and addresses of database tables. It is updated when the data tables are modified, or when a new query needs to be processed by AxleDB.

[1] The translation of SQL queries to AxleDB instructions is currently a manual process.

Data Storage Units and Device Controllers

AxleDB is connected to different sources of data, i.e., SSD, DDR-3, and host. ***i)*** As explained, the software extension of AxleDB is located on the host to initialize the configuration of the query execution, by issuing the query-specific AxleDB instructions. Its physical connection is through PCIe-3 interface, with a maximum throughput of 8 GB/s. For this aim, we use Intellectual Property (IP) cores of Xilinx as the PCI-3 device controller. ***ii)*** SSD is connected through SATA-3 interface, with a maximum throughput of 500 MB/s and is used as a primary storage and shelters the database tables. Furthermore, we use a modified version of Groundhog [138] as the SATA-3 device controller. ***iii)*** DDR-3 is used as the secondary storage, with a maximum throughput of 15 GB/s. It is used to locate the input/output data tables and the temporary tables, e.g., hash tables, during the query processing. Also, we use the Memory Interface Generator (MIG) IP core of Xilinx as the DDR-3 device controller.

The column addresses of the database tables in SSD are generated in the following. First, the CStore is setup as all tables would exist in the host. Then, the data layout, namely, all the starting block addresses, block lengths and address offsets between columns of the schemas are extracted. Next, the columns are copied to SSD storage with respect to the data layout. Although, the starting addresses in the SSD may be different than the host CPU, the data layout is kept the same. Apart from the described mechanism, the SSD does not have a filesystem.

AxleDB Accelerators Unit (AAU)

In this section, we explain the overall organization and interconnection of AxleDB accelerators before presenting the AxleDB's execution model. In Section 3.3, we presents the details of each accelerator.

AxleDB is equipped with a set of hardware accelerators to carry out the query processing primitives, i.e., filter, arithmetic, aggregation, groupby, hash build, hash probe, and sort units. Structurally, each accelerator has ***i)*** an input data port to stream in input data (*din*), ***ii)*** an output data port to

stream out the result data (*dout*), ***iii)*** an input signal to determine the state of the unit (*state*), which is elaborated later in this section, and ***iv)*** a set of inputs to define the functionality of the given accelerator, e.g., to define the filtering qualifiers in the filtering unit. Also, some of them have additional ports to access the temporary data during the processing, i.e., hash tables in the groupby, hash build, and hash probe units, and partially sorted data set in the sort unit. For this aim, ***i)*** the groupby unit has input/output ports (*tin*, *tout*) to read/write the hash table, ***ii)*** the hash build unit has an output port (*tout*) to write into the hash table, ***iii)*** the hash probe unit has an input port (*tin*) to read the hash table, and ***iv)*** the sort unit has input/output ports (*tin*, *tout*) to access the partially sorted data set. Other accelerators, i.e., filter, arithmetic, and aggregation units are inherently on-the-fly operations and do not need any access to the temporary data during the query processing.

To interface AAU with data storage units to transfer the input/output and temporary data set, the accelerators are connected through two distinct data buses, with respect to their sequential and random data access types. More specifically, ***i)*** the input/output data are usually accessed sequentially. Thus, the potential long latencies can be covered by streaming data in a pipelined schema. Accordingly, to access the input/output data, we made the design decision to provide a flexible schema. In contrast, ***ii)***to access the temporary data, we set a shortcut path to DDR-3 memory with a minimum latency. Since the temporary data, i.e., hash table and partially sorted data, are accessed randomly, thus, a low-latency path would be efficient. We elaborate the interconnections, as below:

1. ***RingBus of AxleDB Accelerators (RBAA):*** In order to build a flexible AxleDB substrate, we connect the accelerators with a unidirectional ring interconnect that is RBAA. We use the RBAA to only stream in/out the input/output data tables, and not temporary data, from/to the data storage units, i.e., SSD, DDR-3, and host, in a flexible schema. As it can be seen in Figure 1, the accelerators are chained to each other with a specific order, as follows: filter, arithmetic, aggregation, groupby, hash build, hash probe, and sort units. The order follows a typical

DBMS engine processing pipeline order [75]. To set up the chain, we connect the *dout* port of the earlier unit to the *din* port of the latter unit, consecutively. Also, the *din* port of the filter unit and the *dout* port of the sort unit are used to externally interface the RBAA. Also, currently, we design RBAA as a single-channel bus, which leads to a single stream of data in the ring, at a time. Accordingly, to process an SQL query, we need to break it into a set of data streaming paths. Processing of the corresponding SQL query can be accomplished by streaming data through the data paths in a sequential order through the single channel RBAA bus. The elaboration and example query will be presented in Sections 3.2.

2. ***DirectBus of AxleDB Accelerators (DBAA):*** As mentioned earlier, groupby, hash build, hash probe, and sort units needs to be connected to an off-chip storage to access their temporary data set. For this aim, we use a dedicated data bus that we term DBAA. As it can be seen in Figure 1, the *tin* and/or *tout* ports of accelerators are connected to this data bus. It is worth noting that accessing random data set, e.g., hash tables and partially sorted data set, under a long latency would cause a significant throughput degradation. Although, some techniques such as multithreading [46] can alleviate this issue, in the current version of AxleDB, our sort and hash-based accelerators are single-threaded. Alternatively, in AxleDB, we make some design decisions to decrease the latency of accessing the temporary data, by dedicating a direct data bus, as:

    - We believe that DDR-3 is the only promising accommodation among the available data storage units, i.e., SSD, DDR-3, and host, to cope with the temporary data. DDR-3 has the shortest latency, which justifies it for random data accesses such as hash tables and partially sorted data set.
    - Accessing DDR-3 through the ring bus incurs an additional latency (at max 7 cycles for each data access), which as explained, may cause performance degradation. Thus, to access the temporary

data, we dedicate the direct bus, so-called DBAA without any additional latency. Similar to the RBAA, the DBAA is also a single-channel bus, which corresponds to transfer a single set of data, at a time.

In summary, we equip AxleDB with ***i)*** RBAA to provide flexibility to access the input and output data tables from various data storage units, and ***ii)*** DBAA to quick access to the DDR-3 for some of the costly query operations, i.e., group by, hash build, hash probe, and sort units. The inherent advantage of having two distinct data buses is maximizing the overall throughput when various storage units are exploited for each data bus. For instance, by using SSD for the RBAA and DDR-3 for the DBAA, the bandwidth of both SSD and DDR-3, can be utilized, at a time. On the other hand, assuming that the RBAA is also connected to the DDR-3, the bandwidth of the memory needs to be shared among RBAA and DBAA. To share the bandwidth of DDR-3, we would need an arbiter that serializes the concurrent DDR-3 requests. The arbiter is a part of Programmable Interconnection Unit (PIU), which we give more details on Section 3.1.3.

Nevertheless, according to the specific accelerated SQL query, a subset of the hardware accelerators are usually utilized at a time and not all of them. To meet this requirement in the hardwired chain structure of the RBAA, we design the accelerators to work in two distinct states: *active* or *silent*, which can be set by using an input signal. In the *active* state, the accelerators normally work, as expected to carry out the expected function of the query primitive, e.g., filtering, aggregating. In contrast, in the *silent* state, the accelerators work as bypass buffers and only pass the incoming data to the next unit in the ring (with a single-cycle latency), without applying any function. As a consequence of this structure, depending on the SQL queries, we can utilize only the required subset of the accelerators, by setting them to the *active* state and by setting other accelerators to the *silent* state. Considering the architecture, to access data, the chain incurs a latency of maximum 7 cycles. However, as mentioned earlier, we use it only for streaming input and output data tables and not for temporary data, which does not

cause any throughput degradation, since the additional latency is covered by streaming the sequential data access.

Note that the structure of the AAU may constrain supporting the SQL queries or achieving a fully optimal query plan of a given query. Below, we discuss the possible constrains of AxleDB:

1. ***RBAA as a Hardwired Chain of the Accelerators:*** Although, the accelerators are chained inside a hardwired ring, the current order is viable enough to make an efficient query plan. Performing a filtering operation at the beginning can significantly reduce the size of the data for the next costly operations, i.e., group by, hash build, hash probe, and sort units, as they need frequent accesses to the off-chip DDR-3. In summary, by following this design point, the off-chip data accesses can be decreased, which in turn, corresponds to a significant throughput increase. We show more detail on this design point in Section 3.2.

2. ***DBAA as a Single-channel Data Bus:*** In the DBBA, only a single stream of data can use the bus, at a time. In other words, at a time, one of the corresponding accelerators, i.e., group by, hash build, hash probe, and sort, can be active. Consequently, in an AxleDB-specific query plan for a given SQL query, we need to be ensured that each sub-query exploits only one of the aforementioned hardware units to access random temporary data in DDR-3. We elaborate the query planning of AxleDB for an example query in Section 3.2.

Programmable Interconnection Unit (PIU)

PIU is designed to make AxleDB flexible enough to exchange the data among SSD, DDR-3, host, and RBAA. Also, it synchronizes the data movement among different clock domains and manages the bandwidth sharing. The PIU is composed of the following components:

- A 4-port Programmable Data Connection Switch (PDCS) to exchange the data blocks among SSD, DDR-3, host, and RBAA. To support all the possible data movement cases among the data sources, its ports are

Table 3.1: AxleDB instructions for query execution

Query Execution Instructions (QEI)	
Instruction	**Description**
filter#	a generic filtering
arith#	an arithmetic op.
aggreagate#	an aggregate op.
groupby#	hash-based groupby
hash_build#	building the hash table
hash_probe#	probing the hash table
sort#	sorting data
minmax_index#	index checking

designed to be bidirectional, whereas each direction of each port can be utilized independently.

- The buffering and synchronizing FIFO modules to manage the data movement among the various ports of the PDCS. For the data transfer, buffering and synchronization is a crucial mechanism because, first, various data sources have different latencies, and second, they work with different clock frequencies. For each read and write direction, separate FIFOs are dedicated.

- An arbiter to manage the DDR-3 read/write requests, as DDR-3 has two distinct access modes, first, a direct connection from some of the accelerators in the DBAA to access their temporary data and second, an indirect connection through the PDCS to access the input/out data tables. In the arbiter, among the aforementioned concurrent DDR-3 requests, we set the higher priority to the requests from direct connection DBAA to quickly serve the requests for the temporary data.

In summary, PIU is designed to efficiently share the bandwidth of the data storage units, while it can provide a fully flexible data movement schema.

Data and Process Controller (DPC) and AxleDB Instruction Set

DPC orchestrates the involved modules of AxleDB to process the complex SQL queries in a fast, efficient, and flexible schema. More specifically, it

Table 3.2: AxleDB instructions for data movement

Data Movement Instructions (DMI)	
Instruction	**Description**
HOST-SSD#	stream data between host and SSD
HOST-DDR#	stream data between host and DDR
SSD-DDR#	stream data between SSD and DDR
HOST-DPC#	send instructions from host to DPC

manages the movement of data and the activation of hardware accelerators to execute an SQL query, by issuing the control signals to the PIU and the AAU, respectively. Accordingly, the control signals from DPC, ***i)*** to the PIU determine the source and the destination of the data movement, and ***ii)*** to the AAU activate those accelerators that need to be utilized in the RBAA for any certain SQL query. Also, the parameters of each accelerator are determined by DPC, e.g., the filtering qualifiers for the filter unit.

The control signals are used by DPC to manage the query processing. They are generated based on instructions of AxleDB. The query-specific AxleDB instructions are translated from the SQL queries in the host. Translation of the SQL queries to the instruction set of AxleDB is currently a manual process. We introduce instructions of AxleDB later in this section. As it can be seen in Figure 1, instructions are located in a shared location in the host memory that is called DAT. Then, they are sent to the IC that resides inside the DPC. The DPC fetches them from the IC, decodes, and executes by the execution FSM. Consequently, the control signals are issued and sent to the PIU and the AAU. Accordingly, the query execution is completed, when all the instructions inside the IC are consumed. In the end, the results of the query can be either stored back in the SSD or sent to the host. The DPC can now synchronize with the host and wait for more instructions to process a new query.

Tables 3.1 and 3.2 summarize the instruction set of AxleDB for performing the tasks of data movement and query execution. Data Movement Instructions (DMI) set up the PDCS to exchange data blocks among the different sources of AxleDB, i.e., SSD, DDR-3, and host, bidirectionally. Furthermore, depending

on the query plan, Query Execution Instructions (QEI) configure AxleDB to activate the corresponding accelerators in the RBAA to start streaming the input data. QEI consists of filtering, arithmetic, aggregation, group by, hash probe, hash build, and sort instructions, as well as MinMax index creation/deletion instructions. Consequently, the available instructions can cover a large subset of the SQL queries by executing them on hardware. However, the unsupported operations can be fallback to the host if they can be scheduled as the last step in the query scheduling stage. DMI and QEI include parameters, e.g., source, destination, key columns, carried column (payload), as well as accelerator-specific parameters, e.g., filtering operations ($<, >, <>$) and its qualifiers for the filtering unit.

3.1.2 The Execution Model of AxleDB

To alleviate the common restrictions of classical processor-based systems, the execution model of AxleDB relies on the streaming of the data through the processing units. For this aim, FPGAs provide a unique opportunity, since their programmable logic blocks that are called LookUp Tables (LUT) can be chained together to construct deep pipelines. In this model, each processing node in the pipeline can be enabled, whenever the inputs are available.

To process an individual SQL query, we use AxleDB instructions to establish the required data streaming paths. In other words, in AxleDB, each SQL query is defined by a set of the data streaming paths. Source and destination of data, e.g., SSD, DDR-3, host, together with the required processing units in the AAU constitute a data streaming path. Accordingly, to process an SQL query, we need to make a query plan by breaking the SQL query to a set of sub-queries which are suitable for the AxleDB's components that we have presented in the previous section. The generated sub-queries are one-by-one mapped to data streaming paths that can be run in AxleDB. Later on, we generate the required instructions and configure AxleDB to establish the corresponding data paths, sequentially. And finally, by streaming the data through the established data paths, the processing of the query can be accomplished. In the current version of AxleDB, at a time, we can establish

a single data streaming path. This property leads to a sequential, in order and one by one, execution model for the data streaming paths, which lead to having single stream of data in components of AxleDB, at a time. As it can be seen, first, the DPC is initialized by a set of AxleDB instructions that are copied from DAT in the host to the IC in the FPGA (A). Depending on the type of each valid instruction (B):

- DMI control to exchange data among SSD, DDR-3, and host. Thus, after appropriately configuring the PDCS to set up the required source, destination, and direction, (C) the data are streamed in (F).

- QEI controls the hardware accelerators, then stream the payload for AAU access. In the AAU, the corresponding accelerators are activated, and others are configured only to pass the data (E). Furthermore, for those QEI that need to transmit data to/from SSD, DDR-3, or host, which is determined by the parameters of the instructions, configuring the PDCS is also needed (D). Finally, data streaming is started through the established data path. As mentioned, at a time, we have a single stream of data in components of AxleDB. Thus, before reading new instruction, the current stream needs to terminate executing (F). Later on, we proceed to read the next instruction from IC (B) to start making the next data streaming path.

This process continues until consuming all the instructions of the IC while updating DAT with new instructions can restart the execution process of AxleDB.

3.2 Illustrating the Execution Model of AxleDB by an Example Query

In this section, to demonstrate how AxleDB works, we illustrate the query processing procedure for an example query. The example query is Q03 from TPC-H benchmark [127] which is typically used to test data analytics

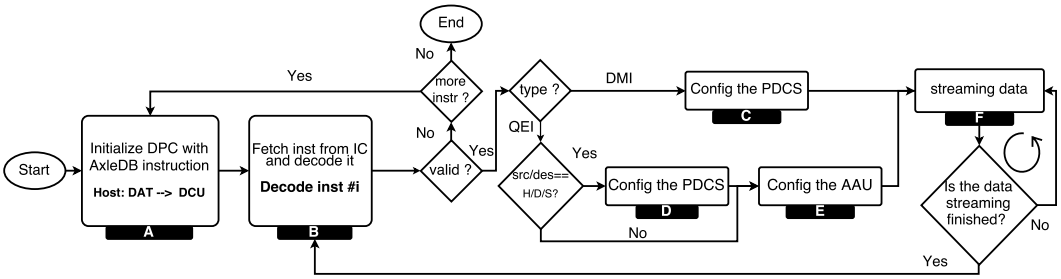


Figure 3.2: The flowchart of the execution model in AxleDB. (In this figure: H= Host, S= SSD, and D= DDR-3.)

performance. AxleDB runs this query without any required modifications or code rewriting, where most of the accelerators presented in this work are utilized. Since AxleDB is designed to be programmable, we can follow many different query plans to execute the queries. However, in this example, to show a comprehensive execution model of AxleDB, where input data is located in the SSD, we built a customized query plan. Accordingly, we first load the input data from SSD to the DDR-3, and then, start query execution by retrieving data from DDR-3. In the rest of this section, we start by introducing an example query. Later on, show how AxleDB processes this example query by describing an optimal query plan, by introducing the list of the required AxleDB instructions to run the query plan, and by explaining how these instructions program the components of AxleDB to utilize the required modules.

3.2.1 Elaborating the Example Query

The example query is shown in the Figure 3.3 (a). In a typical SQL query, several language elements such as **SELECT FROM**, **WHERE**, **GROUP BY**, and **ORDER BY** can exist. These operations can be semantically mapped to specialized hardware accelerators. In this example, the **SELECT** statement fetches the desired data columns (*l_orderkey*, *revenue*, *o_orderdate* and *o_shippriority*) **FROM** the given tables (*customer*, *orders* and *lineitem*). The **WHERE** statement is used to restrict the data in the tables and includes operations such as logical comparisons and arithmetic operations

that filter the data relevant to the user (e.g. *o_ orderdate* < date *'1995-03-17'*, *l_ shipdate* > date *'1995-03-17'* and *c_ mktsegment* = *'FURNITURE'*). AxleDB's filtering units can be exploited to handle the **WHERE** clauses of SQL queries. When multiple tables are involved, the JOIN statement (*c_ custkey* = *o_ custkey* and *l_ orderkey* = *o_ orderkey*) is used to combine the data in these tables, based on a common field (*custkey* and *orderkey*). This operation can be efficiently mapped to AxleDB's hash join accelerator. The **GROUP BY** statement aggregates data into groups based on a given field (i.e. *l_ orderkey, o_ orderdate, o_ shippriority*), which is mapped to AxleDB's hash-based groupby accelerator. The **ORDER BY** statement sorts the data in ascending or descending order based on a given key (i.e. *o_ orderdate, revenue*), which can be performed using AxleDB's sorter accelerator. The **LIMIT** statement causes to fetch a limited number of records.

3.2.2 How does AxleDB Process the Example Query?

To process an SQL query in AxleDB; first, the host generates a set of DMI and QEI. Currently, this is a manual process, but it can be automated by following a similar approach with Glacier [89]. Figure 3.3 (b) shows a simplified diagram for one possible query plan for processing the query Q03 on AxleDB, where the mapping from the key operations of the query to AxleDB's accelerators is indicated by letters (from A to F). To have a simpler figure, we did not show many details of AxleDB, i.e., RBAA, PDCS, instruction parameters, etc. Also, as it can be seen, the execution is accomplished in nine distinct steps. Each step is distinguished by using a set of arrows with its unique numbers and colors in the figure. The label of each arrow shows the key columns that are used in the corresponding part of the processing. Within each step, data can be streamed in a pipelined fashion, and between steps, it is sequential (in order and one by one), due to data dependencies.

Due to column-oriented data store format of AxleDB, only the 10 required columns out of a total of 33 columns (of three input data tables) are loaded from SSD to AxleDB. Moreover, while loading data from SSD to DDR-3 memory, the columns of data are converted into batches that are appropriate

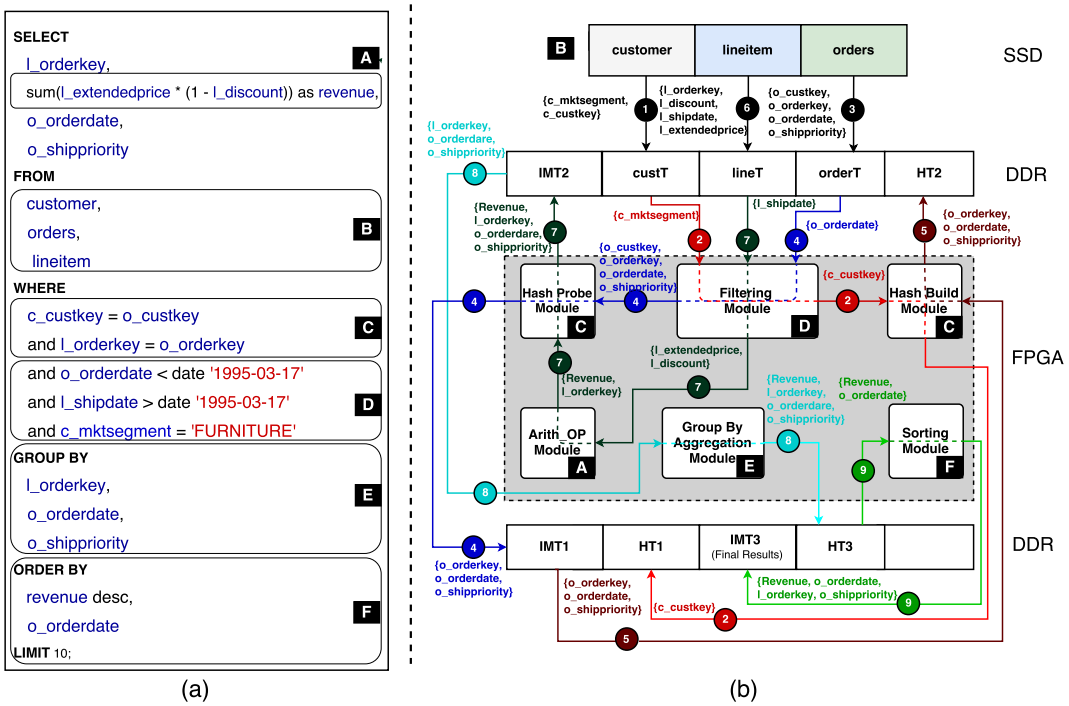


Figure 3.3: (a) Example SQL query: Q03, (b) An example query plan of AxleDB to process Q03. To have a simpler figure, ***i)*** we partition the tables of the DDR-3 memory into two boxes at above and below of the FPGA, although, AxleDB is currently attached to a single channel of DDR-3, and ***ii)*** we only show essential fields of the labels of arrows, excluding the input parameters of accelerators, payloads, etc.

for the processing units. Also, during the processing, different types of data can be stored in the DDR-3 memory, i.e., input data tables (custT, ordersT, lineT), intermediate data tables (IMT1, IMT2, IMT3), and temporary data tables, e.g., hash tables (HT1, HT2, and HT3). To access to the aforementioned data tables from the DDR-3, the memory bandwidth is shared. Previously explained, we manage bandwidth sharing by using an arbiter to serialize the concurrent memory requests. We elaborate it in Section 3.3 for a sample data path, step #7 as below, of the example query. In a nutshell, we perform the following steps to run Q03 on AxleDB (The presented numbers are for the 1GB scale of the dataset. However, more information about our benchmark environment is presented in the following section):

1. Query processing starts by loading only the necessary columns of *customer* table to DDR-3, using 'DMI: SSD-DDR#'. *c_mktsegment* and *c_custkey* columns are loaded to DDR-3 and others are skipped.

2. For *customer* table, first performing a filter on *c_mktsegment*, using 'QEI: filter#' reduces size of data from ≈ 150K to ≈ 30K records. Later on, for the filtered data, a hash table (HT1) is built into the DDR-3 based on *c_custkey* field, using 'QEI: hash_build#'.

3. Query processing resumes by loading only the necessary columns of *orders* table to DDR-3, using 'DMI: SSD-DDR#'. *o_custkey*, *o_orderkey*, *o_orderdate* and *o_shippriority* columns are loaded to DDR-3, and the others are skipped

4. For *orders* table, first performing a filter on *o_orderdate*, using 'QEI: filter#', reduces the size of dataset from ≈ 1.5M to ≈ 725K records. Later on, HT1 is looked up based on *c_custkey*, using 'QEI: hash_probe#'. The resulting joint table is stored into the DDR-3 (IMT1) with ≈ 145K records of data.

5. Applying a nested hash join process, IMT1 is used as the input table to build the second hash table based on *o_orderkey*, using 'QEI: hash_build#'. The hash table is stored into HT2.

6. Query processing continues by loading the necessary columns of *lineitem* table, using 'DMI: SSD-DDR#'. *l_ orderkey*, *l_ extendedprice*, *l_ doscount* and *l_ shipdate* columns are loaded to DDR-3, and others are skipped.

7. For *lineitem* table, first performing a filter on *l_ shipdate*, using 'QEI: filter#', reduces the size of the input dataset from ≈ 6M to ≈ 3.2M records. Later on, in a pipelined fashion the arithmetic unit is exploited to compute *revenue*, using 'QEI: arith#'. Probing the second hash table (HT2) based on *l_ orderkey*, using "QEI: hash_probe#", it generates the resulting joint table into IMT2, with about 30K records of data.

8. In this step, data records of IMT2 are grouped into the new table (HT3), based on a merged *key* (*o_ orderkey*, *o_ orderdate*, *o_ shippriority*). In addition, an aggregation on *revenue* field is performed, using 'QEI: groupby#'. A total number of groups (records) is ≈ 11K in HT3.

9. The query processing is finalized by sorting all groups of HT3 based on a merged *key* (*revenue*, *o_ orderdate*), using 'QEI: sort#', and transferring top 10 records to PostgreSQL, using 'DMI: DDR-HOST#'. (The host is not shown in the diagram, for a clearer figure.) The final result in IMT3 can also be written into the SSD, using 'DMI:SSD-DDR#'.

Due to explained query plan of Q03, the required instruction set of AxleDB to process the given query is summarized in Table 3.3. They are composed of many parameters, e.g., the operation (that defines the appropriate operation), the source (to determine the source of the data and corresponding address), the destination (to determine the destination of the data and the corresponding address), the key columns (the columns that are used as key during the query execution) and the payload (the columns that are only carried along). In summary, these instructions are used to establish the required data streaming paths in AxleDB to process the example query Q03, by following the query plan in Figure 3.3. Accordingly, through a sample example, it is illustrated how these instructions are used to establish one of the sample data streaming paths, #7, by following the process in Figure 3.2.

Table 3.3: AxleDB instructions to process the example query, according to the query plan in Figure 3.3 (some of the fields such accelerator-specific parameters are omitted in this table.) The size column shows the size of the data stream for each data path, in terms of the number of rows, in 1 GB scale dataset.

#path	**#instr**	**Instruction Fields**					
		operation	**src**	**dest**	**key_cols**	**payload**	**size (1GB)**
1	1	SSD-DDR	customer (SSD)	custT (DDR)	c_mksegment, c_custkey	—-	150K
2	2	filter	custT (DDR)	—	c_mksegment	c_custkey	
	3	hash_build	—	HT1 (DDR)	c_custkey	—-	30K
3	4	SSD-DDR	orders (SSD)	orderT (DDR)	o_custkey, o_orderkey, o_orderdate, o_shippriority	—-	1.5M
4	5	filter	orderT (DDR)	—	o_orderdate	o_custkey, o_orderkey, o_shippriority	
	6	hash_probe	—	IMT1 (DDR)	o_custkey	o_orderkey, o_orderdate, o_shippriority	0.72M
5	7	hash_build	IMT1 (DDR)	HT2 (DDR)	o_orderkey	o_orderdate, o_shippriority	0.14M
6	8	SSD-DDR	lineitem (SSD)	lineT (DDR)	l_orderkey, l_discount, l_shipdate, l_extendedproce	—-	6M
7	9	filter	lineT (DDR)	—	l_shipdate	l_orderkey, l_discount, l_extendedproce	
	10	arith	—	—	l_extendedprice, l_discount	l_orderkey	30K
	11	hash_probe	—	IMT2 (DDR)	l_orderkey	Revenue, o_orderdate, o_shippriority	
8	12	groupby	IMT2 (DDR)	HT3 (DDR)	l_orderkey, o_orderdata, o_shippriority	Revenue	11K
9	13	sort	HT3 (DDR)	IMT3 (DDR)	revenue, o_orderdate	l_orderkey, o_shippriority	11K

3.2.3 Establishing a Data Streaming Path: Elaboration for a Sample Data Path

In this section, we explain how components of AxleDB are leveraged to process the example query. As it can be seen in Table 3.3, the processing of Q03 can be accomplished by 13 AxleDB instructions that lead to creating 9 distinct data streaming paths. Among them, and as an example, we explain the required steps to create the sample data streaming path #7, which is depicted in Figure 3.4. As it can be seen, to establish the given data path #7, 3 QEI are used (#9, #10, and #11). Each QEI determines a specific part of the data path #7 and finally, by the last instruction the path is established. Due to each instruction, the DPC generates the necessary control signals to the PIU, to set up the data movement path, and to the AAU, to activate and configure the required hardware accelerators. Accordingly, to establish the given data path, steps as below are proceeded:

1. ***Instruction #9:*** As it can be seen in Figure 3.4(a), instruction #9 defines a filtering operation for the data in the DDR-3. Thus, the required actions are ***i)*** configuring the PDCS to stream in the data from the DDR-3, in this case lineT, to the AAU-RBAA. For this aim, the appropriate ports of the PDCS are utilized. And, ***ii)*** activating the corresponding hardware accelerator, in this case filter unit, by setting it to work in the *active* state (state=1). Also, the query-specific filtering parameters are defined by DPC, "*l_ shipdate > 1995-03-10*".

2. ***Instruction #10:*** As it can be seen in Figure 3.4(b), instruction #10 defines an arithmetic operation for the filtered data. Thus, the only required action is to activate the arithmetic accelerator by setting it to work in the *active* state (state=1). Also, the convenient parameters of the arithmetic unit need to be set. For this aim, the DPC generates the required control signals to carry out the corresponding arithmetic operation, "*l_ extendedprice*(1-l_ discount)*". It is worth noting that for this instruction, as it does not have any valid source or destination parameters, we do not need to modify the PDCS configuration.

3. ***Instruction #11:*** As it can be seen in Figure 3.4(c), instruction #11 completes the establishing of the data path #7, as it needs to write data into one of the data sources, DDR-3. Thus, data streaming will be started after this final step. Instruction #11 requires utilizing the hash probe accelerator by setting its state to *active* (state=1). Consequently, the other accelerators in the AAU including aggregation, group by, hash build, and sort units are configured to work in the *silent* state to only pass the incoming stream of data to the next unit in the RBAA. Also, the hash probe key, in this case *l_ orderkey*, is determined by the DPC. The hash probe unit needs another DDR-3 memory access to read the hash table, in this case HT2. For this aim, the corresponding port of the DBAA is activated to access the hash table from DDR-3.

In summary, the data streaming path #7 leads to ***i)*** read the lineT table from DDR-3, ***ii)*** filter the incoming data stream based on *l_ shipdate* item, ***iii)*** apply an arithmetic function for the filtered data, ***iv)*** probe the stream of the data based on the *l_ orderkey* in the HT2 hash table., and finally, ***v)*** write the probed data into the DDR-3 in the index IMT2. In total, there are 3 distinct DDR-3 data access paths in this data path, i.e., ***i)*** to stream in the input data (as explained above in the part of instruction #9) through RBAA, ***ii)*** to access the hash table in the hash probe unit (as explained above in the part of instruction #11) through a direct DBAA bus, and finally, ***iii)*** to stream out the result data (as explained above in the part of instruction #11) through RBAA. To manage the concurrent DDR-3 requests and optimally share its bandwidth, the arbiter in the PIU is exploited, which allows the priority to the DBAA requests to quickly serve the hash table accesses.

3.3 Query Processing Accelerators

In this section, we go through the architecture of the proposed accelerators that are implemented to perform efficient query processing in AxleDB. We proposed query execution units including filtering, arithmetic, aggregation, sorting, hash join, and groupby, as well as the MinMax indexing mechanism

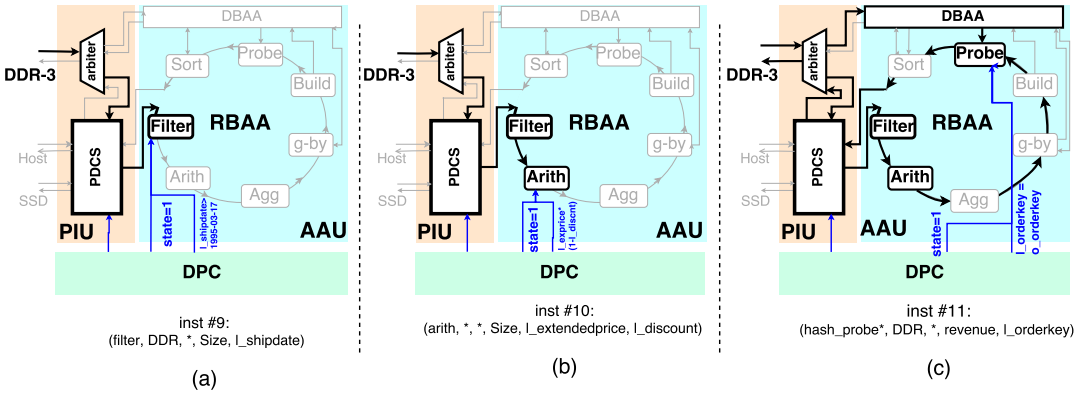


Figure 3.4: Establishing the data streaming path #7 by composing together different AxleDB instructions #9, #10, and #11. The detailed connections between DPC, AAU, and PIU are shown. Among them, the highlighted components/connections represent the corresponding parts that are utilized by each AxleDB instruction to set up the given data streaming path. The control signals for the PIU and the AAU are generated by DPC.

for I/O optimization. In the rest of the chapter, we assumed input data table as a set of tuples, pairs of *key* and *value*. *key* refers to the column(s) of data, used for performing the main query operation, e.g., sorting key in a sorter unit. *Value* refers to the other columns that need to be carried to make the final resulting data.

For developing, leveraging HLS tools, we have designed the filtering and aggregation accelerators in Vivado HLS [145], where we have been able to exploit the data parallelism via Vivado's compiler directives. For task-parallel and control-oriented accelerators, such as the hash join and sort engines, Bluespec SystemVerilog [23] is used. Verilog RTL code is employed for the integration of interface controllers. We select to employ these HLS tools as a result of our previous empirical analysis of a representative set of HLS tools for database acceleration [10].

3.3.1 Filtering Operations, Arithmetic, and Logic Unit

Database filtering is relational operations that test the numerical or logical relations between columns, in the form of numerical and/or Boolean values. Arithmetic unit handles addition, subtraction, multiplication and division operations. Logic unit is designed to handle logical AND, OR and NOT operations.

The key importance of filtering operations in an SQL query is to reduce the amount of data for further processing [78], [79]. For this purpose, we designed a compile-time parameterizable, variable width, n-way compute engine that takes in rows of data as inputs, applies a filtering operation to the desired fields and produces an output bitmap. This bitmap determines the resulting rows for further processing. Similarly, we designed arithmetic, and logical compute engines. The arithmetic engine supports the integer Add, Sub, and Mult operations. We omit the floating point division operator for its large FPGA area requirements. In our experiments, we needed the division operator in a single query (Q14). However, in general, in similar scenarios, the division operation can be handled at the host. Logical operations of NOT, AND, OR, NOR and NAND are also supported. Also, keywords such as IN, SOME, and EXISTS can also be mapped to multiple logical operations. The design behind the filtering, arithmetic and logical blocks encapsulates three major decisions:

– **Width of *key*:** In this thesis, we explored 32-bit and 64-bit data widths for filtering, arithmetic, and logical operations. There are no limitations regarding custom data width selection; since Vivado HLS supports it. Nevertheless, using larger data widths means utilizing more LUT resources. This becomes specifically critical for low-cost FPGAs because they include LUTs with fewer inputs. Overprovisioning data widths can result in area utilization problems and decreased computational power by failing to meet the timing constraints.

– **Number of parallel units:** The number of units determine how much data-parallelism can be supported. For this purpose, the approach we followed is to determine the data widths according to the width of DDR-3 RAM line.

Thus multiple blocks can process a memory line that is composed of multiple elements of data. In this work, one line of DDR-3 RAM is 512-bits and data widths could be either 32 or 64-bits. Thus, 512/32 =16 or 512/64 = 8 units are instantiated in parallel. It is worth noting that in the TPC-H benchmarks that we looked into, we haven't hit to the cases where the required data width is more than 512 bits. Since the 512-bit data width is a property of the DDR-3 interface itself, for more data width, the requirements would be to either (i) use a newer/different technology (High Bandwidth Memory(HBM), 3D stacking, hybrid memory cube, etc.) that supports a wider memory interface, or to (ii) lay the data out in parallel DDR-3 channels. In either case, the AxleDB architecture does not have any inherent limitations regarding the bandwidth to memory.

– **Pipelining:** We designed all supported filtering, arithmetic and logical operations of AxleDB to be fully pipelined, with an initiation interval of 1 cycle. Thus, all query plans that allow pipelining can be fully supported by our filtering, arithmetic logical and aggregation blocks.

For a given filtering, arithmetic or logical operation, each input data can either be compared with another input data from another table, or it can be compared with a constant. For this purpose, scratchpad registers (SPR) are utilized. These registers hold values and allow the aforementioned operators to be applied to the input data and the SPRs. Our filter unit is capable of performing numerous logical operations, including the BETWEEN operator. As an example, in Figure 3.5(a), the input data array holds the column values that require filtering. SPR_0 and SPR_1 hold the filtering qualifiers, the values that input data must be compared against. Thus, input data and SPRs are forwarded to the filters and output are generated. For instance, to perform date *'1996-01-11'* $<$ *l_shipdate* $<$ date *'1995-01-11'* BETWEEN operation, SPR_0 holds date *'1995-01-10'* and SPR_1 holds date *'1996-01-12'* in 32-bit POSIX time format. For other operations, SPRs work the same way.

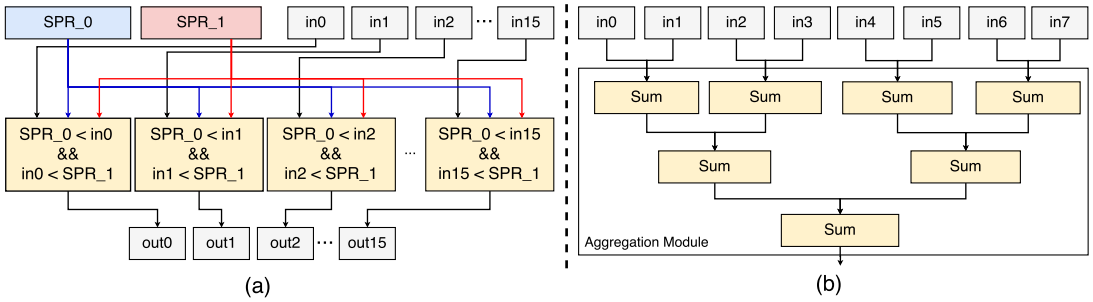


Figure 3.5: (a) Filtering blocks that apply the BETWEEN operation to input data using scratch-pad registers (b) SUM aggregation using binary fan-in technique.

3.3.2 Aggregation Unit

Aggregation operations groups elements and reduce to a single value based on a certain criteria with a more significant meaning, for example mean value of array of elements can be calculated by using sum reduction operator in order to find the total sum. In AxleDB, we provide an n-way aggregator engine that supports MAX, MIN, COUNT and SUM. Aggregation units are designed with the same three design decisions in mind, which were explained previously. They are also fully pipelined with an initiation interval of 1 clock cycle. Similar to the filtering unit, aggregation units are designed to take columns as inputs and to finally combine the results to calculate the final aggregate value. All aggregation operations are implemented in Vivado HLS using the binary fan-in technique. Based on the input array, the size n binary fan-in depth is log2n, and n-1 operators are necessary to form the operator tree. An example is presented in Figure 3.5(b), where for 8 input elements, 7 sum operators are instantiated to generate a single pipelined result.

Multiple filtering/aggregation blocks can be exploited in two ways: pipelining or time-multiplexing. In a pipelined design, streaming data through multiple instances of the accelerators achieve multiple operations in a single pass. However, for area efficient designs, a single accelerator can be used in a time-multiplexed way. For the studied benchmarks, instantiating multiple filter/aggregation blocks in a pipeline provides the maximum throughput, as we further detail with experiments.

3.3.3 Hash-Based Units: Table Join and GroupBy

For table joins and groupby operations, AxleDB uses an efficient hash-based engine. The join operation combines elements of two or more database tables based on a common element. The groupby operation groups elements of rows based on a common feature.

As a first step for table joins, a hash table is constructed using a hash function over the *key* (Build phase). Later on, once constructed, the entries of the second table are probed against this hash table to generate the resulting joint table (Probe phase). For the groupby operation, building a hash table over the *key* can already result in the desired output data. More details of the hash-join based engine can be found in [111].

3.3.4 Sorting and Merging Unit

To efficiently sort large datasets, Axle DB uses a sorter which is an extension of the spatial sorter [96] and allows to effectively support the LIMIT operation. Furthermore, AxleDB employs a merge-sort tree to merge partially sorted sets to be able to sort large input sets. Spatial sorter hardware of AxleDB is not part of the contributions of the thesis and the details of the implementation can be obtained from [111].

3.3.5 Block-Level MinMax DataBase Indexing Unit

Database indexing is a technique that improves the speed of retrieving the database tables and is widely used in software DBMS [93], [83], [100]. AxleDB uses an indexing technique called MinMax indexing for quicker access and localization a subset of data with the expense of space and memory read/write operations. A detailed analysis of the MinMax indexing can be accessed at [111], it is not in the scope of this thesis.

3.4 Evaluation Methodology

3.4.1 Configuration of the AxleDB

AxleDB is developed on a VC709 FPGA development board with an XC7VX690T FPGA and 4GB of DDR-3 RAM. It accesses a Crucial M4-256GB SSD through a customized version of a SATA-3 controller, based on Groundhog [138]. We used a relatively large block size (512 KB), which can help minimize the SSD overheads and, thus lead to significant improvements in data transfer throughput. AxleDB is directly attached to the host through a high-speed PCIe-3 interface for data/instruction transmission. Our accelerators and DDR-3 RAM controllers run at 200 MHz, PCIe-3 controller at 250 MHz, and SATA-3 at 150 MHz, therefore we used various synchronizing FIFOs for clock domain crossing.

In order to thoroughly evaluate the efficiency of the various components of AxleDB, we ran the benchmarks in two modes: ***(i)*** cold run, where the input datasets are originally located inside the SSD, and ***(ii)*** warm run, without considering the I/O time of SSD, and assuming that the datasets are already loaded in the DDR-3 memory of the platform. Thus, in the warm mode, the total processing time of the queries does not include the time of loading input data tables from SSD to the DDR-3. To better analyze the cold and warm runs, we partitioned the total execution time of the query into three parts: ***(i)*** the I/O time of SSD, i.e. the required time of transferring input datasets from the SSD to DDR-3 memory of AxleDB, ***(ii)*** execution time, i.e. the query execution time of the processing units (accelerators) of AxleDB, and ***(iii)*** the time spent on the other parts, including query planning time[2], the time for PCIe-3 data transfers, and finally the overhead of device controllers. The last portion is negligible for large scales of data.

[2]As the query planning of AxleDB is currently a manual process, thus to have a fair comparison with software DBMS, we extracted the average time of the query planner of MonetDB (8ms for cold and 2ms for warm runs), and used it in this part [80].

3.4.2 Configuration of Comparison Cases: MonetDB, PostgreSQL and CStore

AxleDB is evaluated against the query processing engines of several state-of-the-art software DBMS: ***(i)*** MonetDB 11.21 [87] as a popular column-oriented database system, ***(ii)*** PostgreSQL 9.5 (PGSQL) as a popular object-relational row-oriented database system [98], and ***(iii)*** CStore as the PostgreSQL's column-oriented data store extension [33]. More specifically, it is worth noting that:

- **MonetDB** has several unique features to optimize the I/O and computation, simultaneously: ***(i)*** it is built on a column representation of database relations, ***(ii)*** it has an innovative storage model based on vertical fragmentation, ***(iii)*** it has a CPU-tuned query processing architecture, ***(iv)*** it exclusively tries to use the main memory for the processing, and ***(v)*** it has the capability of running queries in a multi-threaded fashion.

- **PostgreSQL** is intrinsically a row-oriented database system. It is equipped with a wide set of database indexing methods, such as BRIN, B-Tree, etc., that can be used as an appropriate comparison case with the proposed FPGA-based MinMax indexing technique in AxleDB. Also, to get better performance, PostgreSQL is extended with an extra patch to support fixed-decimal data type [99]. Fixed decimal is a fixed precision decimal type which provides a subset of the features of PostgreSQL's built-in NUMERIC type, but with increased performance.

- **CStore** is an extension that enables column-oriented data storage in PostgreSQL. It uses the Optimized Row Columnar (ORC) format, which brings some benefits such as; compression, column-projection, and skips indexes (similar to MinMax/BRIN).

We run MonetDB, PostgreSQL, and CStore on a server, equipped with two E2630 Intel Xeon processors, with a total of 12 cores and 24 threads, running at a maximum frequency of 2.3 GHz. The server is attached to a Crucial M4 SSD

disk (as in the AxleDB) to store database tables. The operating system (OS) is Ubuntu 64-bit 12.04, with kernel version 3.13. To have a fair comparison, we equip the server with the same memory size as AxleDB, 4GB DDR-3. However, we have observed system crashes during PostgreSQL/CStore runs for the 10GB scale benchmarks, which is the consequence of the insufficient system memory. Thus, for this special case, the server is equipped with a larger capacity of memory. We observed that at least 16 GB is enough to accomplish the query processing without system crashing. Consequently, in summary, the host is equipped with 4GB and 16GB for MonetDB and PostgreSQL/CStore experiments, respectively. Furthermore, similar to the AxleDB, software DBMS experiments were also ran in two modes, ***i)*** cold mode, where input data tables are located in the SSD, and ***ii)*** warm mode, where input data tables are already loaded into the DDR-3 memory from the original database storage, SSD. To obtain their execution times, we ran each query twice, consecutively. The first run is in the cold mode. In contrast, the second run is executed using internal buffers, where the data is already located inside the main memory of the server. The second run is in the warm mode.

3.4.3 Introducing the Benchmarks Methodology

AxleDB is evaluated with five decision-support TPC-H queries, under various conditions [127]. The studied queries are Q01, Q03, Q04, Q06, and Q14, which heavily utilize and stress the various hardware accelerators. For instance, Q01 requires several aggregation accelerators, Q03 employs nested hash join and sorter operations, and Q06 heavily utilizes the filtering accelerator. Furthermore, the selected queries represent process-intensive (Q01), I/O-intensive (Q06, Q14) and I/O-process-balanced (Q03, Q04) workloads, which allows us to exhaustively analyze the cold and warm runs of the platforms. This classification is based on running the TPC-H queries in default PostgreSQL, equipped with B-Tree indexing, and on a host machine with large enough memory to prevent memory thrashing.

The TPC-H database generator allows generating the input dataset in

various scales. Using this capability, we analyzed the AxleDB in 1GB and 10GB scales, to evaluate it while dealing with small and large datasets. Regarding the maximum size of the data that can be handled in AxleDB, there are two constraints:

- **The size of the input data tables:** The maximum size of the input data tables that AxleDB can handle is constrained by the size of the data storage (DDR-3 for in-memory- warm runs- case and SSD in other cases- cold runs).
- **The size of the hash table:** Other constraint is the size of the hash table that needs to be fitted the DDR-3 size. This is because our hash join/group-by module uses DDR-3 as the hash table and it does not support yet the extremely large hash tables that their size exceeds the DDR-3 size.

It is worth noting that for the studied queries in the experimental results, the aforementioned limitations were never observed. However, through attaching larger DDR-3 RAM to AxleDB, it can cope with larger scales. In addition, other promising solutions include ***(i)*** supporting multiple disks through the use of daughter-cards on the FPGA board [125], ***(ii)*** applying range partitioning on the database tables [140], or ***(iii)*** having a distributed framework [61].

3.4.4 Introducing the Evaluation Metrics

For each given query, we compare AxleDB against the software baselines in terms of speedup, throughput, and energy dissipation metrics. The speedup shows the relative query processing time of the query, throughput in an absolute metric and shows the amount of the processed data in one second, and the energy is used to compare the relative energy dissipation (power * time) of each query. These metrics are defined in equations 1, 2, 3, and 4:

$$speedup = \frac{query\_processing\_time\_of\_sw\_platforms(sec)}{query\_processing\_time\_of\_AxleDB(sec)} \tag{3.1}$$

$$power\_efficiency = \frac{power\_consumption\_of\_sw\_platforms(watts)}{power\_consumption\_of\_AxleDB(watts)} \tag{3.2}$$

$$energy\_efficiency = speedup * power\_efficiency \tag{3.3}$$

$$throughput(absolute) = \frac{total\_amount\_of\_processed\_data(MB)}{query\_processing\_time(sec)} \tag{3.4}$$

3.5 Experimental Results

In this section, first, we individually evaluated the components of AxleDB. Later on, we presented the experimental results of the queries under test and discussed on their performance, including comparisons with multi-threaded DBMS. Finally, we reported and discussed on the power and energy consumption of AxleDB. Later on, the utilization rate of hardware resources is discussed.

3.5.1 Evaluating Query Accelerators of AxleDB

We individually evaluated the query accelerators of AxleDB, including filtering, aggregation, hash join/groupby, sort, and MinMax indexing. In this thesis, filtering and aggregation methods are presented. However, we present the execution time breakdown of queries for each accelerator in Figure 3.7. It is worth noting that, only the query execution time is included in this

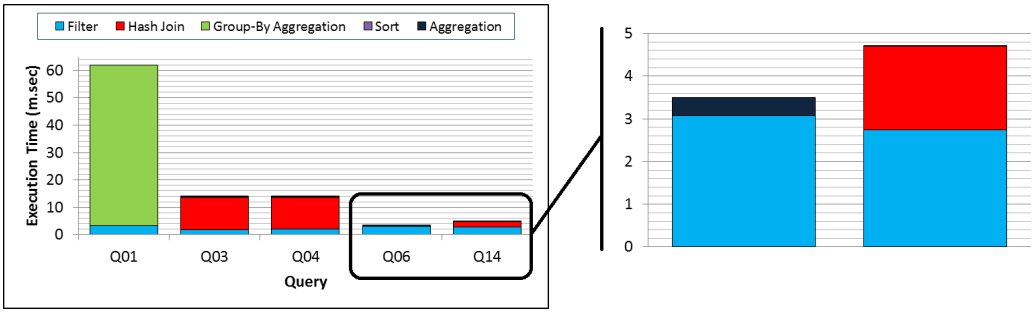


Figure 3.6: The total execution time of the queries, partitioned into per-accelerator

figure, without considering the I/O time of SSD, PCIe-3, or the overhead of controllers.

Evaluating Filtering and Aggregation Accelerators

In the previous section, we presented the design idea behind the filtering, arithmetic, logical, and aggregation units. Before using these units in the AxleDB, we first verified the functionality and also the I/O interfaces, in Vivado HLS [78], [79]. To enable full pipelining, all arrays are completely partitioned using Vivado HLS. To improve performance, the execution latency of all operations take one clock cycle, except for multiplication which takes six clock cycles. For the queries under test, the filtering of char arrays is handled by the filter blocks treating the strings as aggregate 8-bit char arrays. However, for regular expression types of filtering operations, more advanced hardware would be required, which could be incorporated in AxleDB if desired. As it can be seen in Figure 3.6, among the studied queries, the filtering operation is dominant in Q06 and has a significant contribution in Q14, as well. In contrast, for the other queries, as the filtering operation passes a large portion of data to the further expensive operations, its contribution in total execution time is relatively reduced.

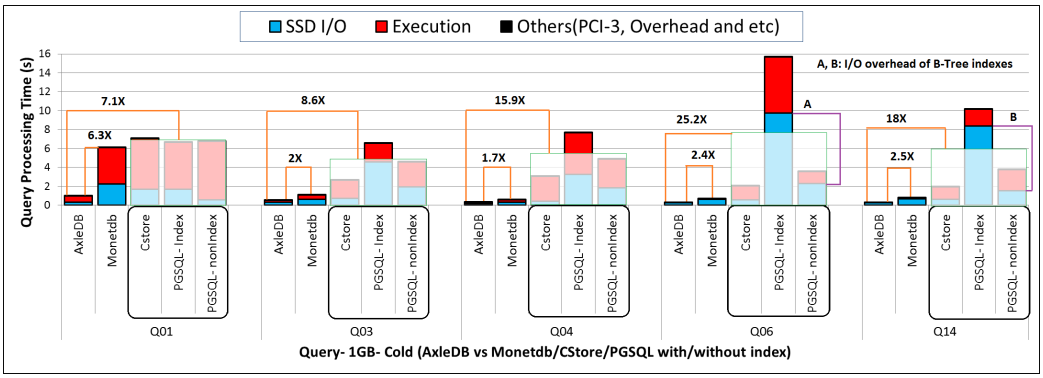
3.5.2 Overall Performance Analysis

In this section, the query processing time of AxleDB is compared against MonetDB, CStore, and PostgreSQL under various conditions. In addition, all the aforementioned accelerators and methods including the join, sorting and indexing operations are incorporated during the experiments. For the overall performance analysis, we ran the platforms in default mode: ***(i)*** MonetDB without any indexing, ***(ii)*** CStore that is equipped with an embedded MinMax indexing (called skip indexing), ***(iii)*** PostgreSQL, which is equipped with B-Tree indexing, as well as an index-free version, and ***(iv)*** AxleDB with the MinMax indexing method. The experimental results of cold and warm runs of the platforms, on 1GB and 10GB scales, are shown in Figures 3.7 and 3.8, respectively.

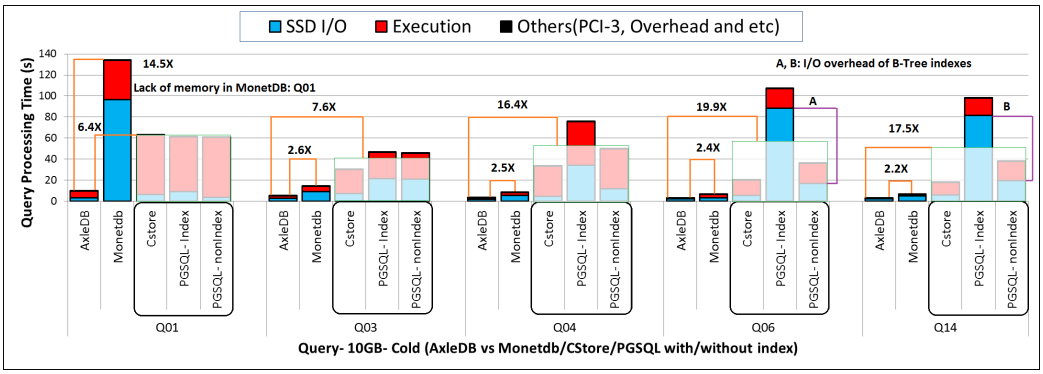
Evaluation of Cold Runs

Figure 3.7 presents the total processing time, broken down into the aforementioned partitions. On average, AxleDB can process queries more than an order of magnitude faster than CStore and PostgreSQL (15x in 1GB and 13.6x in 10GB scales), as well as showing speedup against MonetDB (3x in 1GB and 4.7x in 10GB scales). More specifically, among the set of studied queries, we observed that for:

- **process-intensive queries** (Q01), as it can be seen in Figure 3.7(a), for 1GB scale, the I/O time of SSD is negligible, the indexing method is not utilized well, and the compute time is dominating. In the process-intensive workloads, the performance gain of the AxleDB is mainly the consequence of exploiting highly efficient query accelerators, in a deeply pipelined fashion. For this particular query, the speedup is 6.3x compared to MonetDB, and 7.1x against the different versions of PostgreSQL, including CStore, PostgreSQL-indexed, and PostgreSQL-non-indexed. Although for 10GB scale, as shown in Figure 3.7(b), AxleDB, CStore and PostgreSQL expose similar behaviors, we observed that MonetDB is not optimized very well, as it frequently accesses the SSD to retrieve data. This is the result of insufficient memory to store



(a)



(b)

Figure 3.7: Total query processing time of the studied benchmarks in cold mode, comparing AxleDB vs. MonetDB, CStore, and PostgreSQL. (a)1GB scale, (b) 10GB scale. Lower is better.

temporary data for this particular memory-dependant process-intensive query.

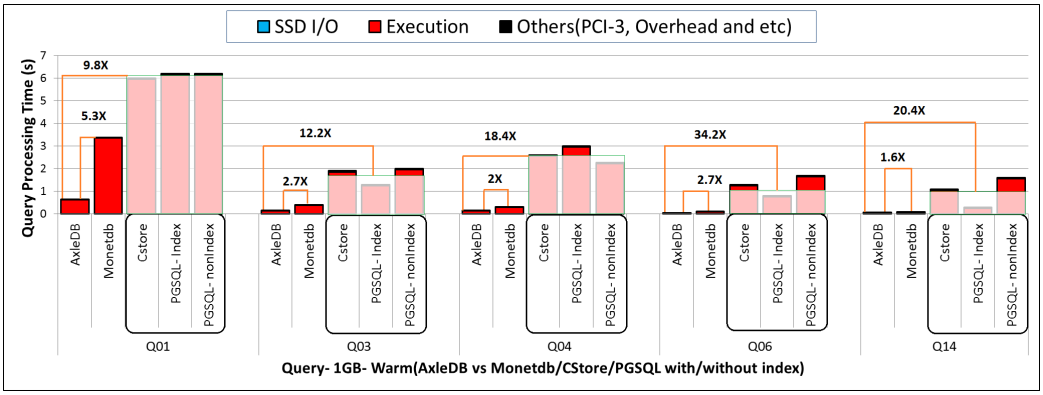
- **I/O-processing-balanced queries** (Q03, Q04), the improvement of AxleDB is the consequence of both I/O efficiency and faster execution. For instance, AxleDB reduces SSD I/O time by 17.9x and execution time by 31.5x for Q04 in 1GB scale, comparing to the index-enabled PostgreSQL, which leads to a total of 23.4x speedup for this particular case. For these queries, on average, the speedup is 1.8x for 1GB and 2.5x for 10GB scale, against MonetDB, and 12.2x for 1GB and 12x for 10GB scale, against different versions of PostgreSQL.

- **I/O-intensive benchmarks** (Q06, Q14), SSD I/O time is dominating. Comparing PostgreSQL with index-enabled vs. non-index versions, we unexpectedly observed a significant overhead of B-Tree indexes, which causes substantial performance degradation. In contrast, AxleDB, CStore, and MonetDB, thanks to their column-oriented data storage, significantly reduce SSD I/O transfers. The results demonstrate that AxleDB can process these queries, on average 2.4x and 2.3x faster than MonetDB, and 21.6x and 18.7x faster than the average of different versions of PostgreSQL, for 1GB and 10GB scales, respectively.

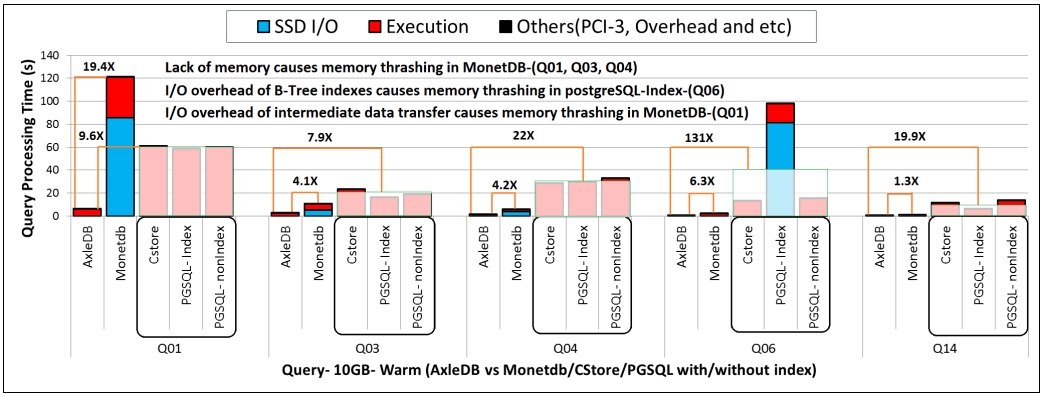
In summary, the results clearly demonstrate that AxleDB achieves the acceleration of query processing, thanks to the pipeline-optimized query accelerators, and simultaneously optimizes the SSD I/O performance, thanks to the data movement techniques that were used, such as direct attached, column-oriented data storage and database indexing.

Evaluation of Warm Runs

For the warm run mode, the speedup of the AxleDB compared to the software platforms is the consequence of the pipelined execution of the query accelerators in the AxleDB. Furthermore, as it can be seen in Figure 3.8(b), in 10GB scale and for some of the queries, the lack of memory in the host



(a)



(b)

Figure 3.8: Total query processing time of the studied benchmarks in warm mode, comparing AxleDB vs. MonetDB, CStore, and PostgreSQL. (a)1GB scale, (b) 10GB scale. Lower is better.

causes memory thrashing that leads to a performance degradation of software platforms. More specifically, for different scales of datasets:

- In 1GB scale, as it can be seen in Figure 3.8(a), we observed that the attached memory of the platforms is large enough to already store the small-sized data. Thus, the SSD I/O is totally skipped. Consequently, the speedup of AxleDB against software platforms is the sole result of a faster query execution in AxleDB. The speedup of AxleDB is from 1.6x to 5.3x against MonetDB (on average 2.9x), and from 9.8x to 34.2x (on average 19x) against the different variants of PostgreSQL.

- In 10GB scale, as it can be seen in Figure 3.8(b), we observed several exceptions, where the memory capacity of the platforms is not sufficient to store entire datasets. ***(i)*** For MonetDB, the SSD I/O time contributes to 71%, 52% and 64% of the total query processing time for the queries Q01, Q03, and Q04, respectively. This I/O overhead is the result of the extra data (parts of the input dataset or temporary data that is needed during the processing) retrieved from SSD. ***(ii)*** On the other hand, for PostgreSQL, we observed memory thrashing for the index-enabled version of Q06. This overhead is the result of a large amount of memory used for the indexes. For this particular case, the I/O contributes to 82% of the total query processing time, which, in turn, executes significantly slower than AxleDB (304x). Eventually, the speedup of AxleDB ranges from 1.3x to 19.4x against MonetDB (on average 7.1x), and from 7.9x to 131x (on average 38.1x) against the different variants of PostgreSQL.

In this section, to evaluate the cold and warm runs, we used a single-threaded version of software platforms and observed a significant improvement using AxleDB. For further investigations, next we analyze their multi-threaded version.

3.5.3 A Discussion on the Optimized points of AxleDB in terms of Data Management and Computational Acceleration

In this section, we analyze the speedups of the warm against cold runs of AxleDB against the comparison cases for the studied queries. In general, the significant speedup of AxleDB against software-based comparison cases is the consequence of two optimization points: ***i)*** offloading the query processing onto the FPGA and following the streamline dataflow execution model and ***ii)*** Optimized accesses to the SSD (tightly coupled to the processing units -accelerators- in the FPGA). Accordingly, we analyze their impact in the speedup of AxleDB for each query individually. Toward this goal, by comparing the experimental results in Figure 3.7(a) (cold runs) and Figure 3.8(a) (warm runs) in 1GB scale, we observe that for:

- **Offloading the query processing onto the FPGA:** To evaluate the impact of this optimization point, we use the experimental results in Figure 3.8(a) for the warm runs. We observe on average 2.9x and 18.9x speedup of AxleDB against MonetDB and PostgreSQL, respectively, in 1GB scale. Since there is no memory thrashing in the 1GB scale of warm runs, we can conclude that these speedups are purely the consequence of the FPGA offloading in AxleDB. Moreover, as it can be seen, MonetDB is more optimized than PostgreSQL especially in Q06 and Q14, which can be the consequence of better memory management (and higher bandwidth utilization) in MonetDB.

- **Performing optimized access to the SSD:** To evaluate the impact of this optimization point, we use the experimental results in Figure 3.7(a) for the cold runs. As it can be seen in this figure, the total query processing time is partitioned into the SSD I/O and the execution time in the processing units. Comparing only the SSD I/O time (blue part), we observe that AxleDB is on average 3.1x and 10.8x more optimized than MonetDB and PostgreSQL, respectively, in 1GB scale. This can be the consequence of the better SSD I/O management in AxleDB thanks

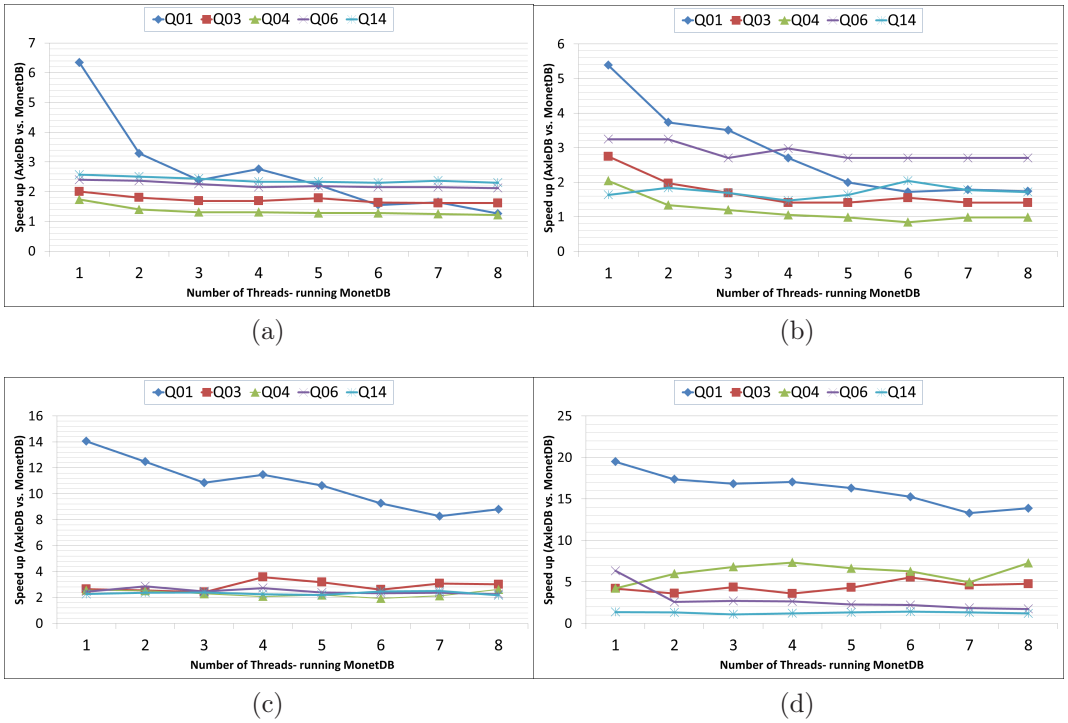


Figure 3.9: Comparing the speedup of AxleDB vs. multi-threaded MonetDB. (a) cold runs in 1GB scale. (b) warm runs in 1GB scale. (c) cold runs in 10GB scale. (d) warm runs in 10GB scale. y-axis represents the speedup of AxleDB against the multi-threaded MonetDB- the relative query processing time, as formulated in the Equation 1. Higher is better.

to the tight coupling of SSD to the hardware accelerators. This lets AxleDB skip the loading of the unnecessary parts of the data tables, and provides a higher utilization rate of the SSD bandwidth by using the large block sizes (up to 1MB). Moreover, as it can be seen, the SSD I/O management in MonetDB and CStore version of PostgreSQL is more optimized than the default version of PostgreSQL, as they follow a column-store data format.

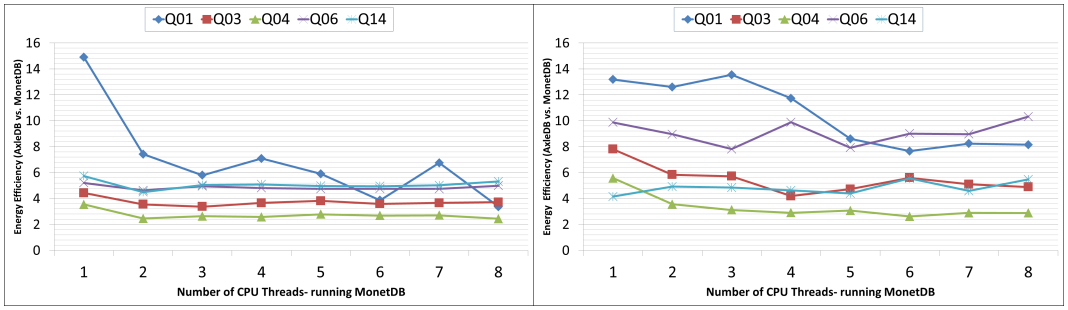
3.5.4 Evaluating the Performance of AxleDB against Multi-threaded MonetDB

To analyze the effects of the multi-threading, we compared AxleDB against the multi-threaded version of MonetDB. Unfortunately, as of now, PostgreSQL and its variants do not support multi-threading. Figure 3.9 shows the experimental results for cold and warm runs of 1GB and 10GB scales, in terms of the normalized speedup of AxleDB vs. MonetDB, exploiting a varying number of CPU threads. We did not observe any significant changes in the behavior of MonetDB utilizing more than eight threads. Thus the diagram includes the experimental results up to eight threads. We observed that:

- For cold runs, as it can be seen in Figure 3.9(a) and (c), utilizing additional threads does not lead to improving the performance of MonetDB, except for Q01, which is a process-intensive query. Consequently, for Q03, Q04, Q06 and Q14, a constant speedup is achieved, almost independently from the number of threads utilized. In contrast, in Q01, utilizing more threads accomplishes the query execution in a parallel and thus in a rapid fashion, which causes to reduce the speedup of AxleDB vs. MonetDB from 6.3x to 1.2x in 1GB, and from 14.1x to 8.8x in 10GB scales. Furthermore, for 10GB scale as a result of memory thrashing issue, in Q01 the speedup of AxleDB is about an order of magnitude, while for the other queries, it is between 1x and 3x.

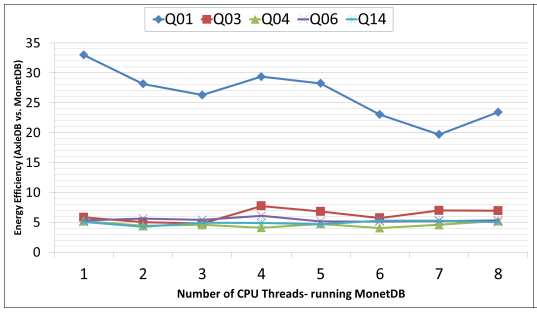
- For warm runs, as it can be seen in Figure 3.9(b) and (d), as there is no SSD I/O transferring time (except those queries where memory thrashing was observed), utilizing more threads leads to better performance of MonetDB. Accordingly, for 10GB scale, the speedup of AxleDB varies between 5.3x and 1.6x, and between 2.8x and 0.98x, against single-threaded and eight-threaded MonetDB, respectively.

In summary, we observed that multi-threading in MonetDB leads to high performance, especially in process-intensive queries. However, as AxleDB simultaneously employs a set of highly-efficient accelerators, as well as optimizing the I/O, it can accomplish the processing of the studied queries faster

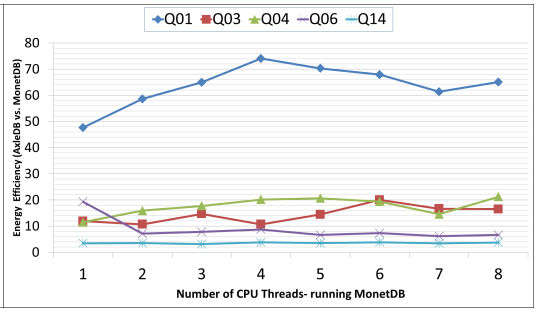


(a)

(b)



(c)



(d)

Figure 3.10: Comparing the energy efficiency of AxleDB vs. multi-threaded MonetDB. (a) cold runs in 1GB scale. (b) warm runs in 1GB scale. (c) cold runs in 10GB scale. (d) warm runs in 10GB scale. y-axis represents the relative energy efficiency of the AxleDB against the multi-threaded MonetDB- the relative energy efficiency as formulated in the Equation 3. Higher is better.

than multi-threaded MonetDB in most cases.

3.5.5 Evaluating the Energy-Efficiency of AxleDB against Multi-threaded MonetDB

Table 3.4 shows the power consumption of AxleDB, estimated using Vivado Power Estimator after the Place & Route stage. A considerable amount of power is dissipated while interfacing hardware and accelerators. Also, multiple clock domains draw additional power in the data connection switch. On the other hand, to measure the power dissipation of the processor, we used Intel's Running Average Power Limit (RAPL) energy meter. RAPL

Table 3.4: Power Consumption of AxleDB components

Component	**Power(W)**
clocks	**1.1**
logic	**0.77**
interfaces	**3.6**
signals	**1.1**
I/O	**1.01**
Acc/PDCS/RBAA DPC/FIFOs	**5.05**
leakage	**0.07**
Total	**12.7**
Total excluding interfaces	**9.1**

exposes energy usage estimates to software via model-specific registers, using hardware performance counters and I/O models [106].

For the comparison against state-of-the-art DBMS, we considered the total power consumption of AxleDB and software platforms, excluding the power dissipation of data storage devices, i.e. SSD and DDR-3 RAM. We reported the energy efficiency of AxleDB against multi-threaded MonetDB in Figure 3.10. It is worth noting that the energy consumption (in Joules) is obtained by measuring the power dissipation using RAPL for software platforms and using Vivado tools for AxleDB (in Watts), and multiplying it with the total query processing time (in seconds). Regarding the experimental results, we observed that:

- As it can be seen in Figure 3.10(a) and (c), for cold runs of MonetDB in 1GB scale, AxleDB is 3.5–14.8x (on average 6.7x), and 2.4–5.3x (on average 3.9x), more energy efficient than the single-threaded and eight-threaded MonetDB, respectively. The improvement for 10GB is more significant, as it varies from 5x to 32.9x (on average 10.8x), and from 4.9x to 23.4x (on average 9.1x). In some cases such as Q01, this is the result of memory thrashing. Furthermore, in cold runs, as the SSD I/O has a significant contribution, the measured power dissipation of

Table 3.5: Hardware Resource Utilization of AxleDB

Component	LUT	FF	BRAM	DSP
filter/aggr/arith	10396	2436	-	256
hash Join engine	13758	10623	724	-
sorter 128-node	148937	131730	-	-
merger (16-to-1)	33061	33840	33	-
index checker	2870	689	45	-
PDCS/RBAA DPC/FIFOs	2401	3884	26	-
SATA-3 ctrl	2018	2607	122	-
DDR-3 ctrl	12226	8329	1	-
PCI-3 ctrl	60671	62993	59	-
Total	283532	256490	1010	256
Virtex-7 usage (%)	65.7	30	68.6	7.1

computing cores does not considerably vary.

- As it can be seen in Figure 3.10(b) and (d), for warm runs of MonetDB in 1GB scale, the energy optimization of AxleDB varies from 5.5x to 13.1x (on average 8.1x), and from 2.8x to 10.3x (on average 6.3x), compared against the single-threaded and eight-threaded MonetDB, respectively. Scaling up to 10GB scale, similar to cold runs, we observed better optimization. Exploiting more threads in warm runs of MonetDB leads to additional power consumption, as more processor cores are active.

Furthermore, compared to the different variants of PostgreSQL, including indexed, non-indexed and CStore, we observed that AxleDB is at least an order of magnitude more energy efficient, on average 25.7x for cold runs and 62.1x for warm runs.

Results show that AxleDB is inherently faster than software DBMS, thanks to its deeply-pipelined architecture, and is also more power-efficient, thanks to a lower operating frequency than software platforms (on average 200 MHz vs. 2.3 GHz), thus as we reported in Figure 3.10, it is more energy efficient, as well.

3.5.6 Hardware Resource Utilization

The flexible design of AxleDB's components allows for various compile time parameterization. For example, the sorter module can be configured for different depths, widths or ascending/descending orderings. Table 3.5 shows the area reports, obtained by setting 16 Bytes for the *key*, and 48 Bytes for the *value* field for the components. This compile time configuration covers all the requirements of the studied queries.

As it can be seen in Table 3.5, we observed that the data-parallel filtering and aggregation accelerators require a significant number of LUTs. The task-based join/groupby modules require an extensive usage of hard memory blocks, because of a high amount of meta-data and the caching circuitry that is generated for providing higher performance. Also, to be able to support all the studied queries, the sorter is configured to be wide enough for running Q01, resulting in a large circuit that occupies around one-third of the FPGA. A 256-node sorter would occupy double this area and would cease to fit our FPGA along with all the other supported modules in AxleDB.

Chapter 4

TauRieL: A Fast Deep Reinforcement Learning Based TSP Solver Using Ordinary Neural Networks

In this chapter, we introduce TauRieL [1]; Deep Reinforcement Learning (DRL) based predictive method. TauRieL complements AxleDB and together, they form the frontend and the backend of the advanced analytics architecture that we have envisioned and defined in Chapter 1.

We present and detail TauRieL's architecture and capabilities by targeting the Traveling Salesman Problem (TSP). Figure 4.1a presents the flow of TauRieL. There exists an agent that is responsible for making decisions by sequentially taking actions. In this setting, taking actions lead to creating traveling salesman tours. There exists a transition matrix which represents the agent's view of the environment by keeping the transition probabilities between different cities. Thus, the agent acts according to the transition matrix, and it determines the agent's policy which is defined as the mapping that indicates which city to travel to next, given a city. When the salesman

[1] A wood-elf character from Hobbit the Movie who possesses *superior senses and pathfinding*

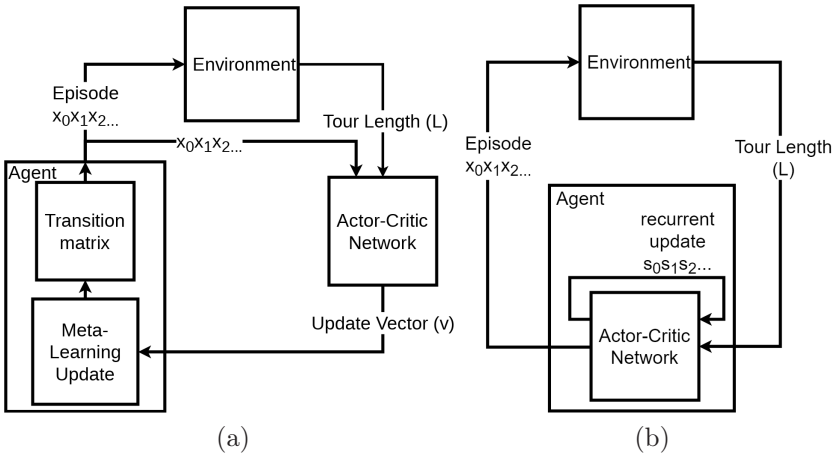


Figure 4.1: High level schemas of TauRieL (a) and state-of-the-art Actor-Critic based TSP solver using RNN [17](b)

completes a valid tour, it is called an episode.

The purpose of the agent is to find the best policy that provides the shortest tour length. At each step, the agent gets a view of the environment, partial or full which is called a state. In order to change states, the agent has to take actions that are fully known by the agent. Thus, the best policy is the optimal mapping from states to actions. Based on the problem or the design decisions, the policy can be either deterministic or stochastic. In the deterministic case, the agent's final policy is a fixed mapping from states to actions whereas in the stochastic scenario, the final policy is the probability distributions all the states and the actions.

TauRieL's agent iteratively updates its view of the environment by generating an update vector v through two neural networks that are inspired by actor-critic reinforcement learning [121]. In deep reinforcement learning context, the actor and the critic are represented by two different neural networks. Moreover, the actor network is the neural net that is responsible for representing the best policy that yields the shortest tour lengths. Furthermore, the critic network is the neural net that represents the state action value which measures how well an agent is doing throughout the episodes. In most cases, it is equal to the expected total reward of an agent starting from an

initial state and continuing all possible permutations [121].

In TauRieL, the actor network is responsible for generating the update vector v that updates the transition matrix, and the final policy is obtained from the transition matrix by choosing the paths with the highest likelihoods. The critic network is responsible for estimating the Euclidean tour length from a given tour, and its primary duty is to improve the training process because the agent's estimation and the rewards obtained from the environment are used together in training.

Moreover, rather than having a fixed update rule, the agent learns to update the transition matrix from v which we call as the meta-learning update step in Figure 4.1a. For large problem sizes, we divide the given input set into a set of clusters and solve the TSP for subsets of points, i.e., sub tours from the given set. Then, the sub tours are merged for forming the global tour using the proposed merging algorithm.

Among available DRL based TSP solvers, TauRieL and NCO [17] may be considered similar in the following sense. Both methods optimize the objective which is to minimize the expected tour length to search for the optimal tour for actor net and estimate the tour length from a given permutation to compare it with the previous searches for critic net. Furthermore, by sampling and exploring the state space, both seek to find the optimal tour. Finally, the neural networks are trained using the REINFORCE algorithm which has been used widely in the reinforcement learning domain in general [121].

TauRieL employs feedforward neural nets as opposed to NCO which [17] employs Ptr-Nets architecture [131] in the reinforcement learning setting as shown in Figure 4.1b. In Figure 4.2 Ptr-Net architecture is introduced. The architecture consists of three major units namely the encoder, the decoder, and the attention. The encoder network is composed of multiple RNN blocks that receive a city per time step in a sequence and generates latent vectors. Each city is a d-dimensional vector that is generated by an embedding network through the linear transformation of inputs. The decoder network is also composed of RNN blocks and also maintains its latent memory states. At each step, uses the attention to produce a distribution over the next city to visit in the tour. Attention network resembles as weighted sum block which

aims to minimize overall objective. After the maximum likely city is selected, the next stage receives the selected city as its input. Thus, the final policy is obtained by inferencing the Ptr-Net by giving a start city and obtaining the output from the decoder stage.

Apart from replacing RNNs with feedforward nets, TauRieL complements the neural nets with the transition matrix. Hence, TauRieL's actor net generates a vector v as opposed to estimating the policy directly by NCO. Then, we use v to improve the state transition matrix from which we obtain the policy. Also, TauRieL performs online training which unifies the training and the search of the shortest tour whereas the state-of-the-art by default requires substantial training duration and dataset that precede the search step. Omitting this step introduces longer training times and poorer performance. Finally, our neural nets take raw inputs, and the design idea behind this decision is to keep neural net sizes relatively small as opposed to the architectures with embeddings such as [131, 17, 65].

The differences above have made TauRieL a more sample efficient architecture. For example in NCO, a single city is represented by two coordinates with each point defined in 128-dimensional embedding space as shown in Figure 4.2 whereas in TauRieL each point is represented with raw 2-D coordinates. Also, the actor RNN in [17] is composed of 128 hidden layers whereas TauRieL's actor DNN has only eight layers. Furthermore, with the addition of the state transition matrix, we can represent the state space much more efficiently compared to [17] and TauRieL can generate results that are two orders of magnitude faster while searching for the shortest tour and perform within 3% of accuracy compared to the state-of-the-art.

Rest of the chapter is organized as follows, Section 4.1 introduces our RL notations and the states the TSP problem. The rest of the Section 4.1.1 explains the building blocks of TauRieL, shown in Figure 4.1a. Section 4.1.2 explains TauRieL's Actor-Critic intrinsics, Section 4.1.3 discusses how episodes are generated from the transition matrix and Section 4.1.4 presents the procedure to update the transition matrix using the update vector v. Section 4.2 introduces the search algorithm and Section 4.3 presents the clustering and the merging algorithm that is responsible for creating subproblems into

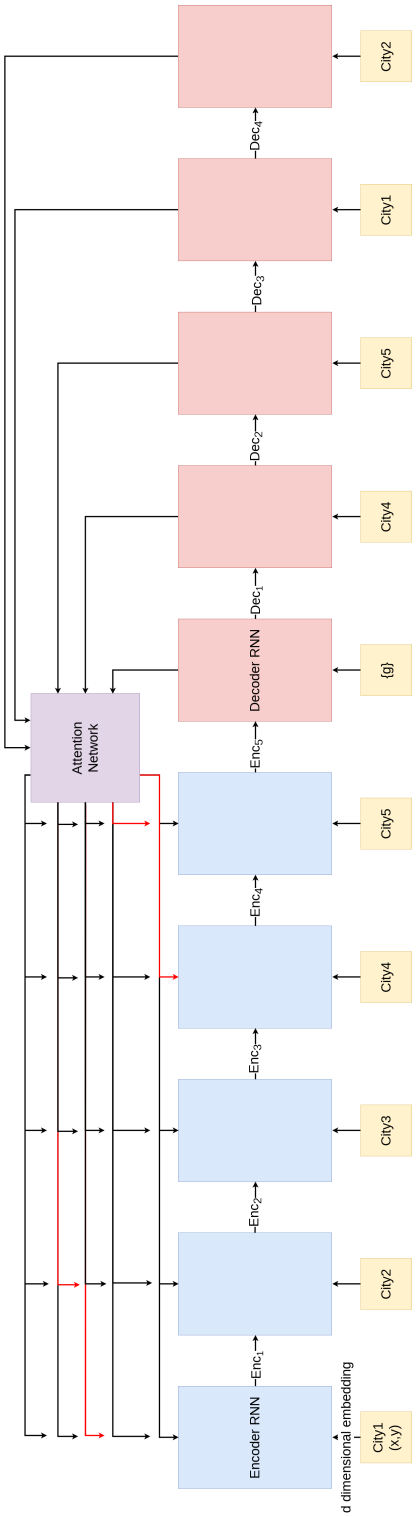


Figure 4.2: Ptr-Nets recurrent architecture [131] with attention that is employed by NCO [17]

clusters and the merging step to create the final solution. The DNN and clustering configurations are discussed in Section 4.4 and Section 4.5 finalizes the chapter by presenting the experimental results.

4.1 Reinforcement Learning Method for TSP

4.1.1 Problem Definition and Notations

We propose a reinforcement learning based method that targets the 2-D symmetric traveling salesman problem. Given a graph G that consists of cities $G = \{x\}_i^n$ such that $x_i \in \mathbb{R}^2$, the objective is to find the permutation that yields a tour by visiting each city [73]. We represent the environment as a Markov Decision Process (MDP), which is a tuple $\langle S, A, P, R, \gamma \rangle$. S defines a state space where each state s consists of a city $x \in \mathbb{R}^2 \wedge x \subseteq G$.

P is a state transition probability matrix and $P_{i,j} = P(S_{t+1} = s_j \mid S_t = s_i)$. R is defined as the expected reward such $R_s = \mathbb{E}[R_{t+1} \mid S_t = s]$ and γ is defined as a discount factor $\gamma \in [0, 1]$. We define the reward $r_{i,j}$ as the negative distance between two cities x_i and x_j. A is defined as a set of actions $\{a_1, a_2 \dots\}$. An example of a set actions can be the different directions of controller movements in a video game [85]. We describe policy as $\pi : S \times A \mapsto S$. For TSP, we assume that the cardinality of the action set is one and this action moves the agent between states which we visualize in Figure 4.3. Therefore for TSP, we assume that each state is synonymous with the action. For example, from state s_0 taking action a_0 will transition to a new state with s_i with probability distribution in $P_{0,i}$. The policy $\pi(a_0 \mid s_s)$ transitions to a new state s_{t+1} according to the dynamics $P(s_{t+1}|s_t, a_0)$ and receives a reward $r(s_{t+1} \mid s_t, a_0)$.

A permutation ϕ i.e. a tour from given the graph G and policy π represents city traversals. Thus, in this work, we interchangeably use permutation and episode to allude to a feasible TSP tour. The environment observes the episode and returns the total reward as the length L of the tour. Thus, given a graph L is defined as:

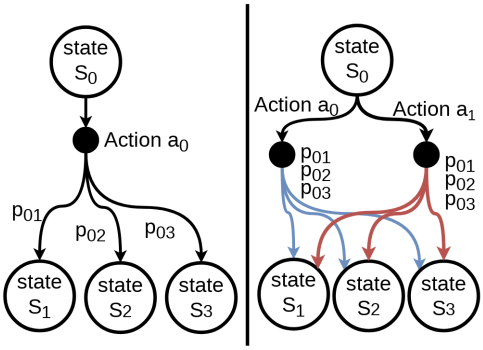


Figure 4.3: In TSP, there exists only a single set of action to move from one state to another (left). There are scenarios where at each state there can be more than one action such as directions of controller movements in a video game (right)

$$L(\phi \mid G) \;=\; \|x_{\phi(n)} - x_{\phi(1)}\|_2 + \sum_{i=1}^{n-1} \|x_{\phi(i)} - x_{\phi(i+1)}\|_2 \tag{4.1}$$

The probability of a tour $p(\phi \mid G)$ describes that the permutations obtained from G have higher probabilities for shorter tours. The probability of a tour can then be shown as the chain rule:

$p(\phi \mid G) = p(\phi_1)p(\phi_2|\phi_1)\ldots p(\phi_{n-1} \mid \phi_{n-2})$.

Whenever a problem can be formulated with a chain rule such as TSP or natural language processing [24], it is an apt candidate to adapt to for recurrent models [120] and more recently sequence-to-sequence models with attention [131, 71]. The advantage of these approaches is that the model output can refer to one of the input elements rather than to a fixed set of reference input such as a language model [144]. Next section introduces our machine learning approach for TSP and presents the formulations.

4.1.2 Using Actor-Critic Reinforcement Learning to Generate the Update Vector

In this section, we present the internals of the Actor-Critic building block of Figure 4.1a which we mentioned in the previous section. The goal is to

optimize the parameters θ_{act} of the neural net that yields the best policy update vector v given permutations ϕ that are generated from the policy π. The parameters of the neural net are optimized with respect to the objective which is the expected tour length:

$$J(\theta_{act}|G) = \mathbb{E}_{\phi \sim p(.|s)} L(\phi \mid G) \tag{4.2}$$

This neural net is called the actor because the gradients with respect to parameters are updated in the direction of improving the update vector v. The gradient $\nabla_{\theta_{act}}$ of the expected tour length is calculated using the REINFORCE algorithm [137] shown below:

$$\nabla_{\theta_{act}} J(\theta_{act}|G) = \mathbb{E}_{\phi \sim p(.|s)} \left[(L(\phi \mid G) - b(G)) \nabla_{\theta_{act}} log\ p(\phi \mid G) \right] \tag{4.3}$$

Using a stochastic batch gradient method, the gradient can be estimated from batches that are sampled from the transition matrix:

$$\nabla_{\theta_{act}} J(\theta_{act}|G) = \frac{1}{B} \sum_{i=1}^{B} \left[(L(\phi \mid G) - b(G)) \nabla_{\theta_{act}} log\ p(\phi \mid G) \right] \tag{4.4}$$

The baseline $b(G)$ is introduced to REINFORCE for reducing the variance of predictions, and using a parametric baseline to estimate the expected tour length were presented to improve learning [121, 17]. However, recent work [128] has shown that specific baselines have been unable to reduce variance for some specialized corner cases. However, this is not the scope of this work.

Therefore, we adopt a neural net that approximates the expected tour length from a given path similar to [17]. Namely the critic, evaluates the current policy by estimating the expected tour length and aims to prescribe towards improved tours [121]. Parameters θ_{cri} of the critic is trained using stochastic gradient descent on a mean squared error objective H between its predictions and the actual tour length from the most recent episode:

$$H(\theta_{cri}|G) = \frac{1}{B} \sum_{i=1}^{B} (b(G) - (L(\phi \mid G))^2 \tag{4.5}$$

Although baseline $b(G)$ is independent of the final policy $L(\phi \mid G)$, the training of the critic network and the actor network occurs concurrently. Both the actor and the critic receive raw input vectors of episodes; the actor outputs the update vector v and the critic outputs a tour length estimation which is represented as the baseline. We detail the meta-learning scheme that updates the transition matrix in the next section.

4.1.3 Sampling from the transition matrix

In this section, we explain how to sample episodes from the transition matrix shown in Figure 4.1a. The sampling of episodes from the transition matrix occurs at each step. Thus the policy $\pi(a_0|S_t = s_i)$ transitions to a new state s_j with probability $P_{i,j}$ in the state transition matrix.

Each row of the state transition matrix $P_{i,:}$ $i \in i, \dots n$ represents a probability distribution. Thus, creating a permutation ϕ from P can be generalized by defining a function which receives a distribution such as P and returns a permutation. In this case for each state, we choose the most likely state from the transition matrix $f(s) = \{\arg\max_p P_{i,:}\}$ and form the episode.

The valid episodes that are fed into the actor-critic and the episodes that are generated from the state transition matrix as shown in the Figure 4.1a.

4.1.4 Learning to update the transition matrix

The actor net in the actor-critic architecture in Figure 4.1a is responsible for producing the update vector v and after each K episode the transition matrix is updated with v. Thus, in this section we explain the transition matrix update procedure which corresponds to the Meta-Learning Update building block in Figure 4.1a.

We treat this step as updating the parameters of a neural net towards the final parameters learned on a task through a gradient descent optimization algorithm [18]. We present the update in Equation 4.6:

$$P_{i,j} = P_{i,j} + \epsilon\,(v_i - P_{i,j}) \quad \forall i\,[i \in 1, \dots, n]\ and\ \exists j\,[j \in 1, \dots, n] \tag{4.6}$$

Algorithm 1 Pseudo algorithm of TauRieL

Input: input graph G, number of search steps *steps*, batch size B, episode samples T, learning rate ϵ, update steps K
Output: the shortest tour length L_{min}, the policy that yields the shortest length π, state transition matrix P
Initialize actor and critic neural net parameters θ_{act} and θ_{cri}
Initialize the transition matrix P
$\phi \leftarrow RandomEpisode(G)$
$L_\pi \leftarrow L(\phi \mid G)$
for t=1,...,*steps* **do**
  $\phi_i \leftarrow SampleEpisodes(P(. \mid S = s_i))$ for $i \in \{1, \ldots, B\}$ (Sample from Transition Matrix given start state)
  $j \leftarrow argmin(L(\phi_1), \ldots, L(\phi_B))$ (Shortest tour)
  if $L_j < L_\pi$ **then**
    $L_\pi \leftarrow L_j$
    $\pi \leftarrow \pi_{\theta_j}$
  end if
  $\nabla_{\theta_{act}} J(\theta_{act}|G) = \frac{1}{B} \sum_{i=1}^{B} [(L(\phi \mid G) - b(G)) \nabla_{\theta_{act}} log\, p(\phi \mid G)]$ (Actor gradient approx. Eqn. 4)
  $H(\theta_{cri}|G) = \frac{1}{B} \sum_{i=1}^{B} (b(G) - (L(\phi \mid G))^2$ (Eqn. 5)
  $\theta \leftarrow RMSProp(\theta_{act}, \nabla_{\theta_{act}} J(\theta_{act}|G))$
  $\theta_{len} \leftarrow RMSProp(\theta_{cri}, \nabla_{\theta_{cri}} H(\theta_{cri}|G))$
  $v \leftarrow p(\phi \mid G)$
  if K steps **then**
    $P_{i,j} = P_{i,j} + \epsilon\, (v_i - P_{i,j})$ (Transition matrix update Eqn. 6)
  end if
end for

The update vector v contains n elements, representing each city in the permutation ϕ. If each element of the permutation is generated from the transition matrix, then each transition $P_{i,j} \forall i \in 1, \ldots, n$ is sampled via $f(P)$ as previously defined. The main design idea of varying the K is to allow sampling from P more than just one step, and it allows more exploration at the current version of the state transition matrix before an update. Additionally, this provides the algorithm to gradually increase K towards the later stages for allowing early exploration. The learning parameter ϵ is a hyperparameter, and we perform a grid search to optimize it [42].

4.2 Unified Training and Searching for the Shortest Tour

The pseudocode for finding the shortest tour is presented in Algorithm 1. The algorithm presents our approach to unified training and searching. Line 3-4 initializes the actor and critic nets and the transition matrix. The transition matrix can either be initialized randomly or from a predetermined initialization that exerts explicit rules. For example, it is possible to prevent certain transitions between states by assigning corresponding probabilities to zero (Line 4). (Line 5) generates a random episode and stores the tour length (Line 6).

The search step starts with sampling episodes from the transition matrix (Line 8). Then the shortest tour is obtained among the samples (Line 9). If the obtained tour is the shortest so far, it is assigned as the current min, and the policy π is updated based on the current minimum tour (Line 10-12). Next, the Actor gradient approximation and the Critic loss are calculated (Line 14-15), and the Actor and Critic nets are forward propagated in this process that is shown with $b(G)$ and $p(\phi \mid G)$. After the backward passes of the actor-critic net (Line 16-17) the transition matrix update occurs after K steps (Line 19-21) using the update vector v (Line 18). The algorithm returns the minimum tour length, the policy that generates the tour lengths and the transition matrix P when the algorithm exists.

4.3 Creating subtours from large graphs

4.3.1 Unsupervised clustering for subtours

Unsupervised clustering analysis is a method for identifying similar points and in the context of TSP where the distance metrix is Euclidian, there has been succesful clustering algorithms [146]. Hence, we have employed K-means clustering [47] for identifying clusters of locations. The objective function we use in the K-means algorithm is to find the best clustering that has the minimum distance between the centroids and the elements. The objective D

is defined as:

$$D = \sum_{i=1}^{N} \sum_{j=1}^{C} d_{ij} |x_i - \mu|^2 \tag{4.7}$$

where x_i represents the n^{th} input point, d_{ij} represents a binary variable to determine which C clusters i^{th} point is located and μ is a vector holding the centroid locations of the C clusters.

We use clustering for two different purposes. The first purpose is to use it as part of the raw input vector for the actor-critic neural RL network. Second, clustering allows to cluster input set into clusters where multiple instances can be executed. Then, the calculated sub tours of each cluster are merged to form the total tour.

4.3.2 Merging sub tours for the total tour

When sub tours are generated by Algorithm 1, we want to merge the sub tours as efficiently as possible, so we developed Algorithm 2. Inspired by the sort-merge join [79], the objective of the merging is to minimize the global tour length. Thus, given the sub tours, the algorithm populates the global tour s_{glob} starting from a predetermined location x_i. The closest element in the sub tours is selected by calculating the distance between the current element in s_{glob} and all sub tour arrays at their current indices, s_{sub} (Line 6). Then, the closest element from a designated sub tour is copied to the global array (Line 7). The selected sub tour array and the global array advance their indices (Line 8). This process iteratively continues until all sub tour arrays are traversed.

4.4 Neural net architecture, clustering configuration, and the input structure

The actor-critic nets consist of two separate multi-layer feed forward neural nets. Each DNN reads identical permutations of cities that are composed of raw inputs. The actor net outputs a vector of v of size n, and the critic net

Algorithm 2 Merge

Input: subtours s_{sub}, starting location x_1
Output: global tour array s_{glob}, global policy Π_{glob}
Initialize global array to starting location $s_{glob}[0] \leftarrow x_1$
Initialize all indices to first elements
repeat
    $s_{glob}[i]$ with all $s_{sub}[j]$ (Calculate the distance between)
    $s_{glob}[i] \leftarrow s_{sub}[j]$ (Assign the subtour with shortest distance)
    $i \leftarrow i + 1$, $j \leftarrow j + 1$ (Advance the global array and selected subtour array)
until all subtours are traversed

outputs a floating point scalar estimating the tour length given a permutation.

For all the layers, we use Relu activation functions. The output logits of the actor net are created using sigmoid activations. During training, both nets are trained using RMSprop [126]. Euclidean distance is used as the distance metric between cities.

All the clustering parameters except the cluster size are predetermined and do not change with the problem at hand. Euclidean distance is used as the distance metric. The maximum number of iterations for the convergence of K-means clustering is selected as 2000 and the starting centroids are selected randomly. For each city i.e. 2-D location $x_i \in \mathbb{R}^2$, it is represented with the input vector I as:

$$I[i] = \begin{bmatrix} x_i \\ Cl_i \end{bmatrix} \tag{4.8}$$

Apart from the raw input locations x_i we concatenate $Cl \in \mathbb{N}$ which represent the cluster location. For example, if there are four predetermined clusters Cl_i can take a value between one and four.

4.5 Experimental results

We present our results in this section, and all the experiments are generated by applying the methods that we explained in the previous sections. We used Tensorflow [3] framework for all the software implementations, and a

Table 4.1: Comparison of average tour lengths using the datasets provided by Ptr-Net [131] and A3 algorithm [1] obtained from [131]

n	Optimal	A3	Ptr-Net	TauRieL
5	2.12	2.12	2.12	2.12
10	2.87	3.07	2.88	2.88
20	3.83	4.24	3.88	3.91
50	N/A	6.46	6.09	6.37

workstation with the following specs are used throughout the experiments: Intel Xeon E5–2630@2.4 GHz, Nvidia K80 and 64GB DDR4@2133MHz RAM. We have experimented with 20 and 50-city instances of TSP and used the dataset from [131] as well as uniformly generated random points $[0, 1] \in \mathbb{R}^2$.

In all the experiments, actor and critic neural nets take mini-batches of 4 instances. For the actor net, we use a 6-layer feed net with:

$[64, 32, 32, 16, 16, number\_of\_cities]$ neurons. For the critic neural net, we use 5-layer feedforward net with $[64, 32, 16, 8, 8, 1]$ neurons. For both nets, ReLu activations are used, and RMSProp training configurations are the following: The learning rates are set to 3e$-$4 and 2e$-$4 for actor and critic, decay is set to 0.96 and epsilon as 1e$-$6. For all cases, state transition matrix is randomly initialized from a uniform distribution $\mathcal{U}(0, 1)$.

The Algorithm 1 requires four hyperparameters; these are the number of iterations *steps*, learning rate ϵ, update steps K for state transition matrix update and sample steps T. In all the presented results, the number of iterations *steps* and the sampling T are set to 250. The learning rate ϵ is set to 0.01. The number of clusters Cl is also predetermined before starting any execution and is determined by the user. In this work, for 20-city instances, the cluster sizes are set to 2 and 5 for respectively, for 20-city and 50-city instances.

We compare our results with a heuristic A3 [1] as well as Ptr-Net [131] and [17] which both use Ptr-Net. Table presents 4.1 average tour lengths for 5 to 50-city instances. For 5-city instances, TauRieL solves optimally and is within 0.05% of the optimal tour for the 10-city. TauRieL outperforms A3

Table 4.2: Execution times in seconds of single episode and sample step for TSP20 and TSP50 instances

	N = 20	N = 50
Graph Gen.	0.014	0.023
Training	0.0034	0.0035
Sampling	0.00123	0.0018
Inference	0.00102	0.0012
Transition Matrix Up.	0.0003	0.0004
Cl Merge	0.007	0.006
Other	0.0005	0.0005

for both 20 and 50 city instances and obtains tour lengths within 0.007 % and 2% of Ptr-Net for 20 and 50-city instances. Apart from 50-city instances, Ptr-Net is trained with optimal results [131]. The 50-city instance is trained with the results obtained from A3 [1]. However, TauRieL started all instances from scratch.

The breakdown of the execution time of TauRieL is crucial, because training, sampling, and inference occur in the main loop. Thus, Table 4.2 presents the execution times of several steps of the algorithm for 20 and 50-city instances. The table allows observing the change of execution times of the steps concerning problem size. Increasing from 20-city to 50-city instances do not affect the training and inferences times significantly as well as the transition matrix update step. In addition, the results show that *Sampling* and *Inference* steps have similar execution time costs. Nevertheless, the sampling step starts to dominate with increasing input.

Similarly, the execution of *Input Gen.* step which is responsible for resizing and appending samples and batches of tensors before starting the search algorithm, increases with increasing data size. The *Sampling* and *CL Merge* costs start to increase when the sample size increases, because at each step of the main loop, the sampling step executes multiple times, and merge step has more sub tours to merge for the global tour. *Others* represents initialization of vectors, the comparison and the update the current best policy and tour length.

Table 4.3: Comparison of execution times and tour length gap from optimal between our implementation and NCO [17]

	Exec. Time (sec) (Train, Inference)	% from Opt.	
n = 20	19860, 0.04	1.4%	NCO
	170	2.6%	TauRieL
n = 50	36021, 0.04	3.5%	NCO
	580	6.1%	TauRieL

Training time is not the dominant factor in TauRieL which is presented in Table 4.3. Also, starting from scratch requires training steps during the lifetime of an algorithm which is the reinforce update shown in Equation 4. We compare TauRieL with Ptr-Net based [17] which applies a similar RL update [137]. We measured the training and inference times of the methods from a reference implementation of [17] a uniformly generated random points $[0, 1] \in \mathbb{R}^2$.

The training times for 20-city and 50-city instances measured from the reference implementation have been 19860 and 36021 seconds respectively. Once the training is finished, the inference is made from the trained model. On the other hand, without needing any data sets, TauRieL runs in 170 seconds for 20-city and 580 seconds for 50-city instances when sample and episodes are both set to 250. Also, for both cases, we obtain 2.4% and 6.1% from the optimal compared to 1.4% and 3.5% which are reported by [17] for 20-city and 50-city instances respectively. Finally, TauRieL does not need any data sets.

In Figure 4.4 and 4.5, we introduce the improvements in tour lengths with respect to the training. For 20-city TSP, the NCO necessitates more than two hours of training in order to perform better than TauRieL which can solve a 20-city instance in less than three minutes. Similarly, for 50-city, NCO needs to train at least eight hours to reach TauRieL's performance whereas TauRieL can obtain a solution in less than ten minutes. Hence, on average training NCO below the specified durations yields poorer results than TauRieL.

Figure 4.4: The average tour length vs training duration for 20-city instances

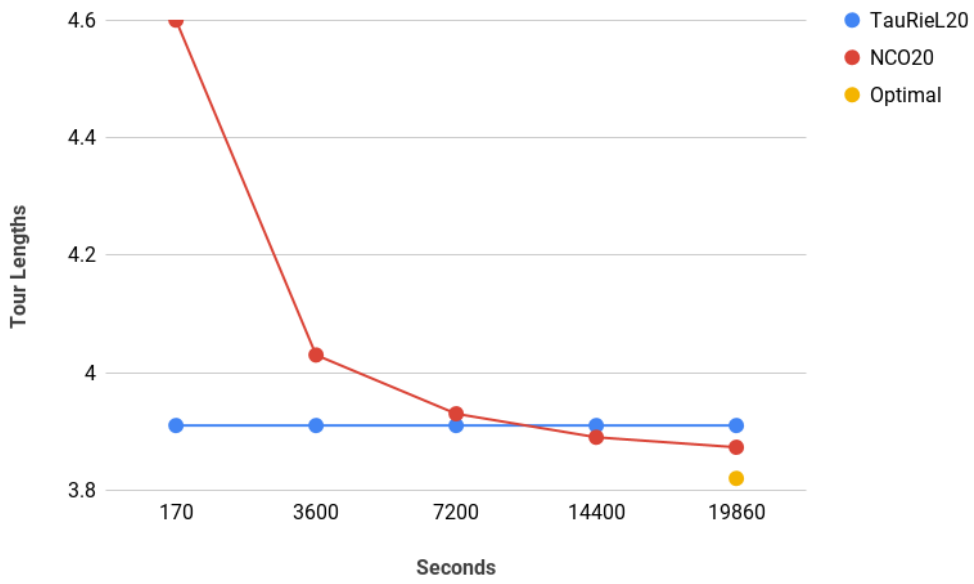


Figure 4.5: The average tour length vs training duration for 50-city instances

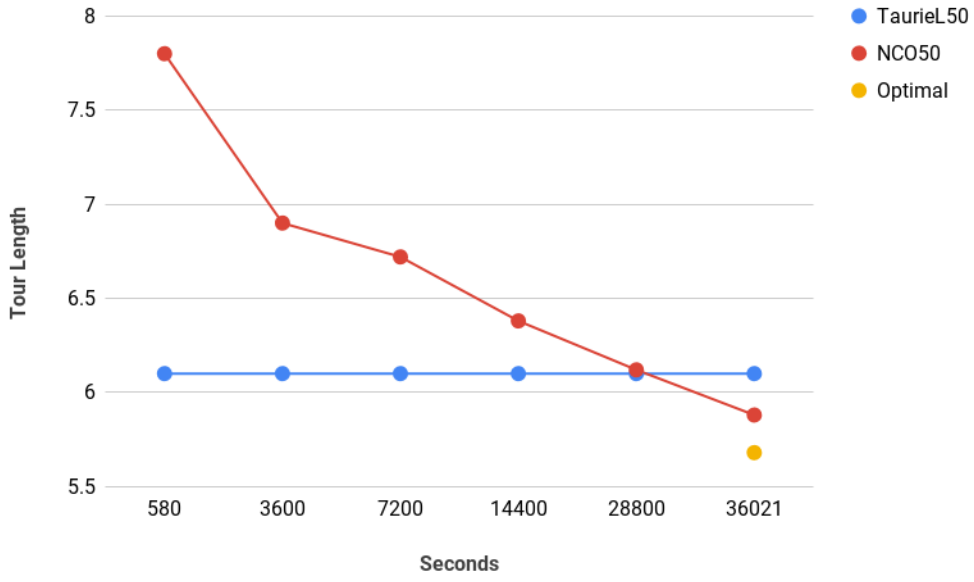
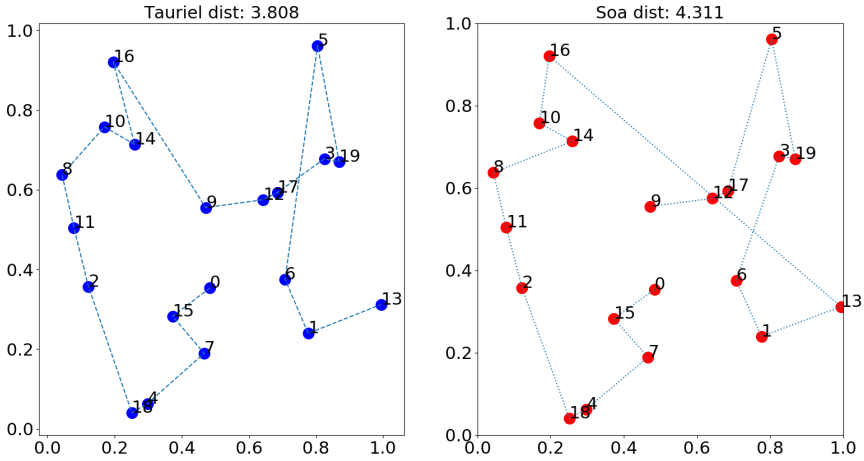


Table 4.4: The % gap from 50-city Ptr-Net [131] with respect to sample size and the number of training steps

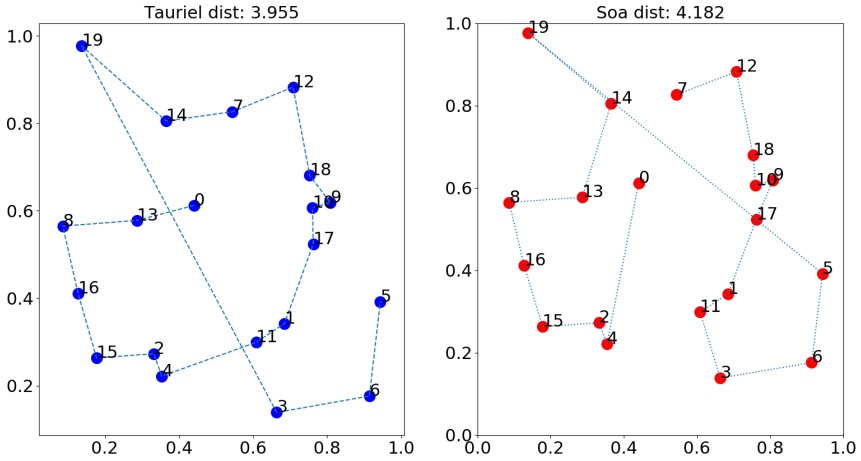
Samples	Training Steps		
	50	150	300
10	18.60	17.40	15.20
50	15.40	10.90	8.40
200	10.10	8.00	7.30
400	8.10	7.90	2.98

The sample size and the number of episodes are the most critical parameters for Algorithm 1. Thus, Table 4.4 displays the change of the tour length gap for the sample size and the number of episodes concerning Ptr-Net [131]. Keeping a low sample size implies fewer explorations in the design space. After 50 episodes, we observe the benefits of increasing the sampling size. Nevertheless, sampling from the transition also has computational costs. Besides, the number of episodes above 150 always have provided the best results. Thus, in order to reach the best results, our strategy has been to increase the sample size and the number of episodes together.

In Figures 4.6 to 4.9, we compare the tour lengths obtained from TauRieL and validation datasets that are used by Ptr-Net and NCO [131, 17]. Figures 4.6 and 4.8 are the examples that TauRieL outperforms the given route, and in Figures 4.7 and 4.9 are the examples that TauRieL underperforms. For both methods, there happen long jumps from nearby dense regions to distant points. Because after touring nearby dense regions the algorithm has had to stochastically continue to another candidate city while still maintaining the nearby dense routes. On the other hand, there also exist counterexamples. In Figure 4.6b, TauRieL has a longer jump between two cities; however overall tour length is shorter because of better routing at the dense regions.

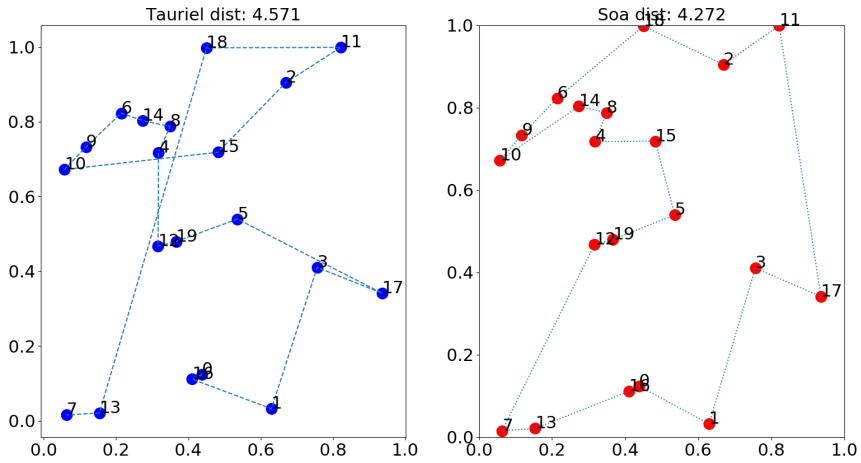


(a)

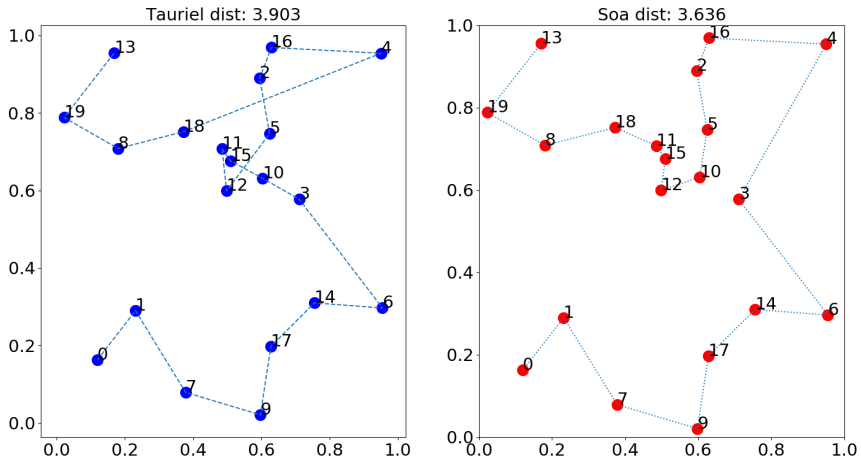


(b)

Figure 4.6: Two example 20-city tour-length results (a) and (b) which perform better than the validation set

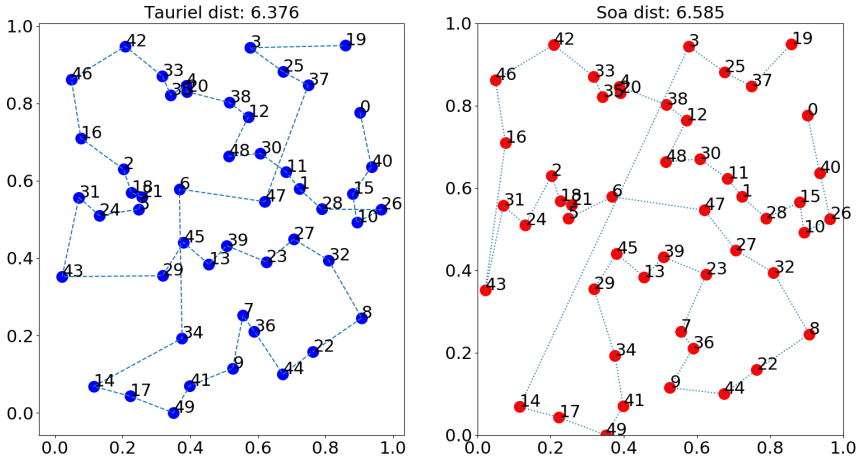


(a)

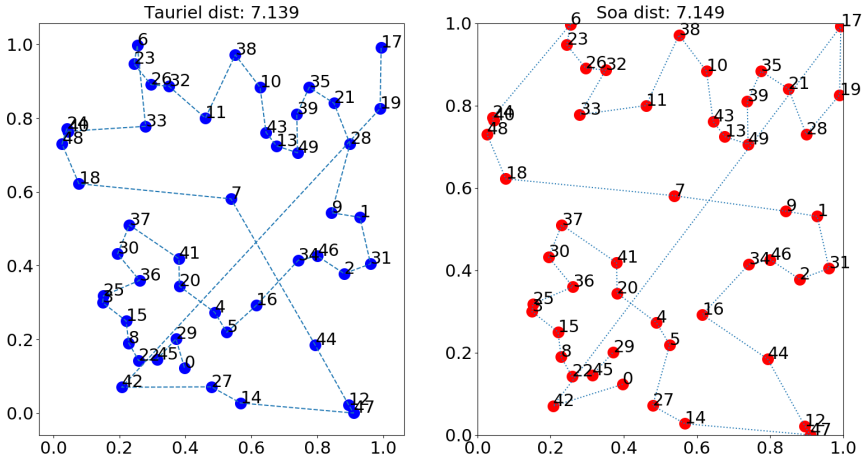


(b)

Figure 4.7: Two example 20-city tour-length results (a) and (b) which perform worse than the validation set

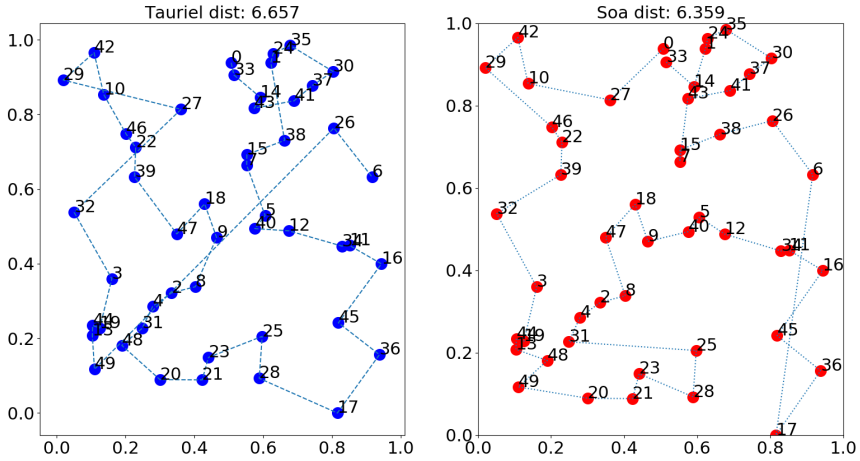


(a)

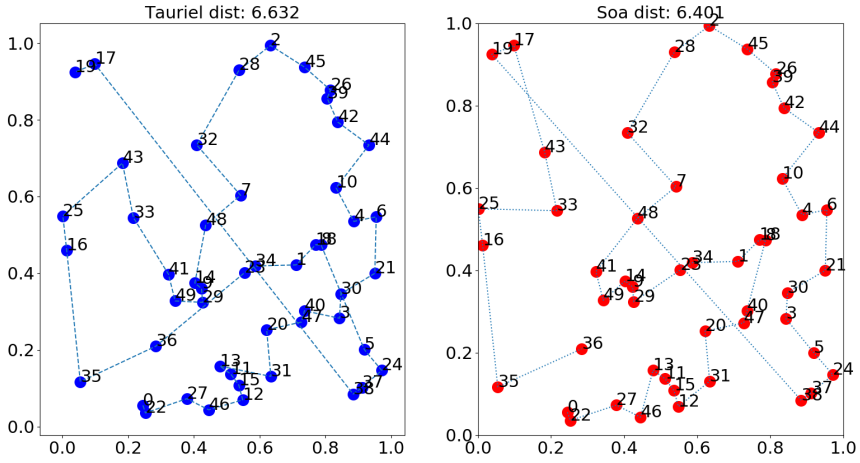


(b)

Figure 4.8: Two example 50-city tour-length results (a) and (b) which perform better than the validation set



(a)



(b)

Figure 4.9: Two example 50-city tour-length results (a) and (b) which perform worse than the validation set

Chapter 5

Adapting TauRieL for Knapsack problem

5.1 Modeling the Knapsack Problem and modifying TauRieL

In this chapter, we present that TauRieL can be adapted for a different combinatorial problem, namely the Knapsack Problem [94]. Given a set of items G, each item κ_i has weight $w_i \in N$ and values $\upsilon_i \in N$ and a knapsack with a maximum allowed weight limit W_{tot}, the problem aims to find the best combination of items with the highest values that do not exceed the knapsack's limit W_{tot}.

$$\begin{aligned} &maximize \sum_{i=1}^{n} \upsilon_i \kappa_i \\ &subject\ to \sum_{i=1}^{n} w_i \kappa_i \leq W_{tot} \\ W_{tot},\ \upsilon,\ w \in \mathbb{R}\ and\ \kappa \in \{0,1\} \quad (i = 1, \ldots, i) \end{aligned} \tag{5.1}$$

The Knapsack problem is also a well known NP-Hard problem as the Traveling Salesman Problem (TSP) [101]. Alternative versions of the problem

have been used in different fields such as satellite management, resource allocation in production management, capital budgeting and cryptography [136]. Numerous exact and heuristic algorithms have been proposed; the earlier proposals have suggested exact solutions using dynamic programming [6] and branch and bound [68] methods. Then, subsequent works have presented relaxation and state reduction techniques [6] for larger problem instances. For enormous problem sizes, approximation heuristics have been proposed. Tabu search [38], particle swarm optimization [13] and evolutionary algorithms [54] are examples of publications of approximation heuristics.

In the Markov Decision Process (MDP) environment $\langle S, A, P, R, \gamma \rangle$ which we have presented in the previous chapter, each state s in the state space S consists of an item b. The transition matrix P is defined similarly as TSP, the transition probabilities of choosing the next state s_{t+1} given a current state s: $P(s_{t+1} \mid s_t)$.

The reward R is defined as the value υ obtained by reaching any state s_i. Specifically, the reward r_i is the value υ_i of item κ_i if it is selected for the sack. A is defined as a set of actions $\{a_1, a_2 \dots\}$ and it is the action to pick up an item κ_j which is represented as s_j in the MDP.

We describe the policy as $\pi : S \times A \mapsto S$. For example, from state s_0 taking action a_0 will be equal to selecting a random item κ_0 we choose another ball $\kappa_i = s_i$ following the probability distribution in $P_{0,i}$. Hence, we are interested in finding the the policy $\pi(a_0 \mid s_s)$ that transitions to a new state s_{t+1} according to the dynamics $P(s_{t+1}|s_t, a_0)$ and generates the highest cumulative reward.

Given a set of items G, an item $\kappa_i = \{w_i, \upsilon_i\}$ possesses the weight and the value of an item. The objective is to find a subset of items $S' \subseteq S$ with the maximum number of values that is less than the total weight W_{top} in which Equation 5.1 presents.

The design idea for adopting the TauRieL for Knapsack is the following. The start state can be selected as either the item with the maximum allowed weight-to-value ratio or a random state among top-k highest weight-to-value states. Then, start searching and picking items one-by-one until they fill the maximum allowed capacity is received (W). Overall, among all the possible

selections choose the maximum. The total reward obtained given item set G can be shown as:

$$L(\phi \mid G) = \sum_{i=1}^{n} v_i \quad n \leq N \tag{5.2}$$

L possesses accumulated values of the subset of the items $n \leq N$ that are selected to let in the knapsack. With this modification, the rest of the RL mechanics that are introduced for TSP in Chapter 4 remains unmodified. The objective which is described in Equation 4.2 is altered into expected item value from the expected tour length by only redefining the total reward in Equation 5.1 in Equation 4.1. The gradient of the objective which is described in Equations 4.3 and 4.4 are used without modifications. Coupled with the new reward function in Equation 5.2, the critic network which evaluates the current policy is estimates the item values given items as opposed to the expected tour length for the TSP. Similarly, the baseline function $b(\phi \mid G)$ that REINFORCE algorithm [121] uses for reducing the variance is defined to be the item value given items.

5.2 Experimental Results

We present the results of Knapsack experiments in this section. Similar to the experiments with TSP, we employ Tensorflow [3] framework for all the software implementations and the same workstation with the following specs: Intel Xeon E5–2630@2.4 GHz, Nvidia K80 and 64GB DDR4@2133MHz RAM. We have experimented with 20, 50 and 100 item instances of Knapsack and used the Knapsack dataset from [131].

In Knapsack experiments, we sweep the number of *steps* from 50 to 10000, and at each step, we keep the sampling T fixed at 250. We report the best solution obtained during the steps. The learning rate ϵ is set to 0.01. We do not apply and use any clustering with Knapsack. We continue to employ two neural nets for actor and critic net. For the actor net, we use a 6-layer feed net with $[64, 32, 32, 16, 16, number\_of\_items]$ neurons.

For the critic neural net, we use 5-layer feed forward net with $[64, 32, 16, 8, 8, 1]$

Table 5.1: Comparison of TauRieL's gap from the optimal versus value-to-weight greedy (vwg) for different problem sizes

(Items, Weight)	Optimal	Tauriel	vwg
(20, 5)	7.82	0.2%	2.95%
(50, 12.5)	20.20	7.28%	14.85%
(100, 25)	40.52	18.8%	24.72%

neurons. For both nets ReLu activations are used and RMSProp training configurations are the following: The learning rates are set to 3e−4 and 2e−4 for actor and critic, decay is set to 0.96 and epsilon as 1e−6. For all cases, state transition matrix is randomly initialized from a uniform distribution $\mathcal{U}(0, 1)$.

We compare our results with the optimal results that we have obtained from Google OR Tools Library [43]. Table presents 4.1 average values for 20 to 100 item instances for optimal, TauRieL and value-to-weight greedy (vwg) heuristic. TauRieL solves 20 item instances within 0.2% of the optimal. TauRieL beats vwg for all the instances. Both TauRieL and vwg deteriorate as the design space enlarges by increasing the number of items.

Table 5.2: The execution times in seconds of running TauRieL from scratch with different item and sample sizes

Items	Sample Steps			
	50	500	5000	10000
20	0.25	2.76	28.3	49.8
50	0.4	4.3	41.2	90.5
100	1.1	8.1	80.0	141.6

We present the overall execution times of 20, 50 and 100-item instances with different step sizes. The exponential state space complexity necessitates more exploration through sampling from the state space [121] as the problem sizes grow. However, the execution times also increases from seconds range to minutes range with increasing state space and samples steps.

The breakdown of the execution time of TauRieL is similar to TSP which we have displayed in Table 4.2. This outcome is expected because the most dominating element in the algorithm are Training and Sampling steps and these steps have remained identical.

Chapter 6

Conclusions and Future Work

In this thesis, we present the contributions to the front and back-end of advanced analytics, namely databases and predictive analytics using machine learning. The ever-increasing data generation and data-oriented predictive methods require efficient and high-performance hardware as well as reductions in data processing time while training machine learning based models.

Hence for the frontend, we present AxleDB, a single node programmable query processing platform that couples efficient query-specific accelerators with memory for providing better performance and energy efficiency compared to state-of-the-art software DBMS. We propose a diverse set of query accelerators for aggregation, filtering, sorting, join and groupby operations to accelerate complex SQL queries. AxleDB is designed to be programmable which enables data movement through memory and storage units and it is programmable from the host CPU by sending instructions through PCI-E. Our experimental results on running a set of queries from TPC-H benchmark suit shows that AxleDB is 1.8x and 34.2x faster and 2.8x to 62.1x more energy efficient compared to the state-of-the-art DBMS, MonetDB, and PostgreSQL that run on a recent server computer.

The results have displayed that AxleDB shows promising results regarding performance and energy efficiency in a single compute node. Based on the results for future work, we imply that scaling the platform into multiple nodes is the reasonable direction. We foresee that extending AxleDB into multiple

nodes will require more effort concerning database operability and networking such as data sharding and bandwidth limitations.

The second contribution in this thesis is TauRieL, a deep reinforcement learning based method that targets combinatorial problems as the backend. TauRieL employs an actor-critic inspired architecture with ordinary DNNs and a state transition matrix. Using TauRieL, we show that we solve 50-city TSP instances in minutes whereas current methods require lengthy training times before inference. Thus, starting from scratch, we can provide results two orders of magnitude faster and within 3% accuracy of the state-of-the-art. With TauRieL, the execution time of DRL based TSP solvers are thus much closer to heuristics based TSP solvers.

Lastly, we have shown how TauRieL handles another combinatorial problem by introducing the Knapsack problem. We first presented how Knapsack can be modeled and solved using TauRieL with minimal changes. Then, we presented the experimental results which compared Knapsack's gap from the optimal while changing the state space and the algorithm's step size. We also compared TauRieL with a highly employed value-to-weight greedy heuristic. Besides, we emphasized how execution times differ with increasing state space and step sizes. We display that up to 100-item problem size, TauRieL could generate results that are closer to the optimal compared to the heuristic.

Currently, heuristics based solutions are still preferable regarding accuracy for larger problem sizes. However, heuristics require a substantial amount of customization for the problem at hand. For future work, we are looking forward to improving TauRieL for tackling larger problem sizes and classes with increased accuracy compared to specialized heuristics.

Bibliography

[1] C++ implementation of traveling salesman problem using christofides and 2-opt, 2017.

[2] Xilinx sdaccel development environment, June 2017.

[3] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[4] Agarwala, R., Applegate, D. L., Maglott, D., Schuler, G. D., and Schäffer, A. A. A fast and scalable radiation hybrid map construction and integration strategy. *Genome Research 10*, 3 (2000), 350–364.

[5] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Wood, D. A. Dbmss on a modern processor: Where does time go? In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK* (1999), no. DIAS-CONF-1999-001, pp. 266–277.

[6] Andonov, R., Poirriez, V., and Rajopadhye, S. Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research 123*, 2 (2000), 394–407.

[7] Anthony, M. *Discrete mathematics of neural networks: selected topics*, vol. 8. Siam, 2001.

[8] Applegate, D. L., Bixby, R. E., Chvátal, V., Cook, W., Espinoza, D. G., Goycoolea, M., and Helsgaun, K. Certification of an optimal tsp tour through 85,900 cities. *Operations Research Letters 37*, 1 (2009), 11–15.

[9] Arcas-Abella, O., Armejach, A., Hayes, T., Malazgirt, G. A., Palomar, O., Salami, B., and Sonmez, N. Hardware acceleration for query processing: Leveraging fpgas, cpus, and memory. *Computing in Science & Engineering 18*, 1 (2016), 80–87.

[10] Arcas-Abella, O., Ndu, G., Sonmez, N., Ghasempour, M., Armejach, A., Navaridas, J., Song, W., Mawer, J., Cristal, A., and Luján, M. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on* (2014), IEEE, pp. 1–8.

[11] Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).

[12] Baldick, R. *Applied optimization: formulation and algorithms for engineering systems*. Cambridge University Press, 2006.

[13] Bansal, J. C., and Deep, K. A modified binary particle swarm optimization for knapsack problems. *Applied Mathematics and Computation 218*, 22 (2012), 11042–11061.

[14] Baraglia, R., Hidalgo, J. I., and Perego, R. A hybrid heuristic for the traveling salesman problem. *IEEE Transactions on evolutionary computation 5*, 6 (2001), 613–622.

[15] Bartholdi III, J. J., Platzman, L. K., Collins, R. L., and Warden III, W. H. A minimal technology routing system for meals on wheels. *Interfaces 13*, 3 (1983), 1–8.

[16] Becher, A., Bauer, F., Ziener, D., and Teich, J. Energy-aware sql query acceleration through fpga-based dynamic partial reconfiguration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on* (2014), IEEE, pp. 1–8.

[17] Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* (2016).

[18] Bengio, Y., Boulanger-Lewandowski, N., and Pascanu, R. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), IEEE, pp. 8624–8628.

[19] Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems* (2007), pp. 153–160.

[20] Bernhard, K., and Vygen, J. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.* (2008).

[21] Bishop, C. M. Pattern recognition. *Machine Learning 128* (2006).

[22] Bland, R., and Shallcross, D. Large traveling salesmen problems arising from experiments in x-ray crystallography: A preliminary report on computation. Tech. rep., Cornell University Operations Research and Industrial Engineering, 1987.

[23] Bluespec. Bluespec systemverilog compiler.

[24] Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. Class-based n-gram models of natural language. *Computational linguistics 18*, 4 (1992), 467–479.

[25] Buchty, R., Heuveline, V., Karl, W., and Weiss, J.-P. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience 24*, 7 (2012).

[26] Carlson, S. When hazy skies are rising. *Scientific American 276* (1997), 106–107.

[27] Casper, J., and Olukotun, K. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays* (2014), ACM, pp. 151–160.

[28] Chen, X.-W., and Lin, X. Big data deep learning: challenges and perspectives. *IEEE access 2* (2014), 514–525.

[29] Christofides, N. Worst-case analysis of a new heuristic for the travelling salesman problem. Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[30] Churiwala, S. *Designing with Xilinx® FPGAs: Using Vivado*. Springer, 2016.

[31] Climer, S., and Zhang, W. Rearrangement clustering: Pitfalls, remedies, and applications. *Journal of Machine Learning Research 7*, Jun (2006), 919–943.

[32] Cover, T., and Hart, P. Nearest neighbor pattern classification. *IEEE transactions on information theory 13*, 1 (1967), 21–27.

[33] CStore-FDW. Postgresql cstore foreign data wrapper extension.

[34] Dennl, C., Ziener, D., and Teich, J. On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library. In *Proc. FCCM, IEEE* (2012), pp. 45–52.

[35] Do, J., Kee, Y.-S., Patel, J. M., Park, C., Park, K., and DeWitt, D. J. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 1221–1230.

[36] Du, D.-Z., and Pardalos, P. M. *Handbook of combinatorial optimization: supplement*, vol. 1. Springer Science & Business Media, 2013.

[37] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (2011), IEEE, pp. 365–376.

[38] Fréville, A. The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research 155*, 1 (2004), 1–21.

[39] Gao, W., Zhan, J., Wang, L., Luo, C., Zheng, D., Ren, R., Zheng, C., Lu, G., Li, J., Cao, Z., et al. Bigdatabench: A dwarf-based big data and ai benchmark suite. *arXiv preprint arXiv:1802.08254* (2018).

[40] Garey, M. R., and Johnson, D. S. *Computers and intractability*, vol. 29. wh freeman New York, 2002.

[41] Glover, F. W., and Kochenberger, G. A. *Handbook of metaheuristics*, vol. 57. Springer Science & Business Media, 2006.

[42] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[43] Google. Google's operations research tools. github.com/google/or-tools, 2018.

[44] Govindaraju, V., Idicula, S., Agrawal, S., Vardarajan, V., Raghavan, A., Wen, J., Balkesen, C., Giannikis, G., Agarwal, N., and Sedlar, E. Big data processing: Scalability with extreme single-node performance. In *Big Data (BigData Congress), 2017 IEEE International Congress on* (2017), IEEE, pp. 129–136.

[45] Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature 538*, 7626 (2016), 471–476.

[46] Halstead, R. J., Absalyamov, I., Najjar, W. A., and Tsotras, V. J. Fpga-based multithreading for in-memory hash joins.

[47] Hartigan, J. A., and Wong, M. A. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics) 28*, 1 (1979), 100–108.

[48] Hassan, A. Increasing performance of in-memory databases using re-ordered query execution plans, Jan. 25 2018. US Patent App. 15/214,082.

[49] Hayes, T., Palomar, O., Unsal, O., Cristal, A., and Valero, M. Vector extensions for decision support dbms acceleration. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on* (2012), IEEE, pp. 166–176.

[50] He, J., Lu, M., and He, B. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proceedings of the VLDB Endowment 6*, 10 (2013), 889–900.

[51] Helsgaun, K. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research 126*, 1 (2000), 106–130.

[52] Hertel, L., Barth, E., Käster, T., and Martinetz, T. Deep convolutional neural networks as generic feature extractors. In *Neural Networks (IJCNN), 2015 International Joint Conference on* (2015), IEEE, pp. 1–4.

[53] Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., Patwary, M., Ali, M., Yang, Y., and Zhou,

Y. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409* (2017).

[54] Hill, R. R., and Hiremath, C. Improving genetic algorithm convergence using problem structure and domain knowledge in multidimensional knapsack problems. *International Journal of Operational Research 1*, 1-2 (2005), 145–159.

[55] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[56] Hölzle, U. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro 30*, 4 (2010).

[57] Hoshen, Y., and Peleg, S. An egocentric look at video photographer identity. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 4284–4292.

[58] IBM. Big data handbook, 2013.

[59] IBM. Ibm netezza data warehouse appliance.

[60] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ACM, pp. 1–12.

[61] Jun, S.-W., Liu, M., Lee, S., Hicks, J., Ankcorn, J., King, M., Xu, S., et al. Bluedbm: an appliance for big data analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ACM, pp. 1–13.

[62] Jun, S.-W., Liu, M., Lee, S., Hicks, J., Ankcorn, J., King, M., Xu, S., et al. Bluedbm: Distributed flash storage for big data analytics. *ACM Transactions on Computer Systems (TOCS) 34*, 3 (2016), 7.

[63] Kaelbling, L. P., Littman, M. L., and Moore, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research 4* (1996), 237–285.

[64] Karp, R. M. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[65] Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems* (2017), pp. 6351–6361.

[66] Koch, D., and Torresen, J. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (2011), ACM, pp. 45–54.

[67] Kolemen, E. *Optimal Configuration of a Planet-finding Mission Consisting of a Telescope and a Constellation of Occulters*. 2008.

[68] Kolesar, P. J. A branch and bound algorithm for the knapsack problem. *Management science 13*, 9 (1967), 723–735.

[69] Kool, W., and Welling, M. Attention solves your tsp. *arXiv preprint arXiv:1803.08475* (2018).

[70] Lawler, E. L. *Combinatorial optimization: networks and matroids*. Courier Corporation, 1976.

[71] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[72] LeCun, Y., Touresky, D., Hinton, G., and Sejnowski, T. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school* (1988), CMU, Pittsburgh, Pa: Morgan Kaufmann, pp. 21–28.

[73] Lin, S., and Kernighan, B. W. An effective heuristic algorithm for the traveling-salesman problem. *Operations research 21*, 2 (1973), 498–516.

[74] Lin, Y., Bian, Z., and Liu, X. Developing a dynamic neighborhood structure for an adaptive hybrid simulated annealing–tabu search algorithm to solve the symmetrical traveling salesman problem. *Applied Soft Computing 49* (2016), 937–952.

[75] Liu, L., and Özsu, M. T. *Encyclopedia of database systems*. Springer, 2018.

[76] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., and Alsaadi, F. E. A survey of deep neural network architectures and their applications. *Neurocomputing 234* (2017), 11–26.

[77] Lv, Z., Song, H., Basanta-Val, P., Steed, A., and Jo, M. Next-generation big data analytics: State of the art, challenges, and future research topics. *IEEE Transactions on Industrial Informatics 13*, 4 (2017), 1891–1899.

[78] Malazgirt, G. A., Sonmez, N., Yurdakul, A., Cristal, A., and Unsal, O. High level synthesis based hardware accelerator design for processing sql queries. In *Proceedings of the 12th FPGAworld Conference 2015* (2015), ACM, pp. 27–32.

[79] Malazgirt, G. A., Sönmez, N., Yurdakul, A., Unsal, O. S., and Cristal, A. Accelerating complete decision support queries through high-level synthesis technology (abstract only). In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015* (2015), p. 277.

[80] Manegold, S. Private communication with monetdb.

[81] McAfee, A., Brynjolfsson, E., et al. Big data: the management revolution. *Harvard business review 90*, 10 (2012), 60–68.

[82] McKinsey, and Company. Big data the next frontier for innovation, competition, and productivity, 2011.

[83] Microsoft. Database indexing in sql server.

[84] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[85] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[86] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[87] monetdb. Monetdb latest released version.

[88] Mueller, R., Teubner, J., and Alonso, G. Glacier: A query-to-hardware compiler. In *SIGMOD '10*, pp. 1159–1162.

[89] Mueller, R., Teubner, J., and Alonso, G. Glacier: a query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 1159–1162.

[90] Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems* (2016), pp. 1054–1062.

[91] Nareyek, A. Choosing search heuristics by non-stationary reinforcement learning. In *Metaheuristics: Computer decision-making*. Springer, 2003, pp. 523–544.

[92] Nguyen, H. D., Yoshihara, I., Yamamori, K., and Yasunaga, M. Implementation of an effective hybrid ga for large-scale traveling salesman problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 37*, 1 (2007), 92–99.

[93] Oracle. Database indexing in oracle.

[94] Papadimitriou, C. H., and Steiglitz, K. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[95] Parashar, A., et al. Triggered instructions: A control paradigm for spatially-programmed architectures. *SIGARCH Comput. Archit. News*, 142–153.

[96] Parashar, A., Pellauer, M., Adler, M., Ahsan, B., Crago, N., Lustig, D., Pavlov, V., Zhai, A., Gambhir, M., Jaleel, A., et al. Triggered instructions: A control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 142–153.

[97] Patterson, D. A., and Hennessy, J. L. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[98] PGSQL. Postgresql latest released version.

[99] PGSQL-Fixeddecimal. Fixeddecimal data type patch for postgresql.

[100] PGSQL-indexing. Database indexing in postgresql.

[101] Pisinger, D. Where are the hard knapsack problems? *Computers & Operations Research 32*, 9 (2005), 2271–2284.

[102] Pohle, T., Pampalk, E., and Widmer, G. Generating similarity-based playlists using traveling salesman algorithms. In *Proceedings of the 8th International Conference on Digital Audio Effects (DAFx-05)* (2005), Citeseer, pp. 220–225.

[103] Pop, D. Machine learning and cloud computing: Survey of distributed and saas solutions. *arXiv preprint arXiv:1603.08767* (2016).

[104] Power, J., Li, Y., Hill, M. D., Patel, J. M., and Wood, D. A. Toward gpus being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (New York, NY, USA, 2015), DaMoN'15, ACM, pp. 11:1–11:8.

[105] Raina, R., Madhavan, A., and Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning* (2009), ACM, pp. 873–880.

[106] RAPL. Intel's running average power limit (rapl) energy meter.

[107] Rosenkrantz, D. J., Stearns, R. E., and Lewis, II, P. M. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing 6*, 3 (1977), 563–581.

[108] Ruan, Z., He, T., Li, B., Zhou, P., and Cong, J. St-accel: A high-level programming platform for streaming applications on fpga.

[109] Russell, R. A. Hybrid heuristics for the vehicle routing problem with time windows. *Transportation science 29*, 2 (1995), 156–166.

[110] Sait, S. M., and Youssef, H. *Iterative computer algorithms with applications in engineering: solving combinatorial optimization problems*. IEEE Computer Society Press, 1999.

[111] Salami, B., Malazgirt, G. A., Arcas-Abella, O., Yurdakul, A., and Sonmez, N. Axledb: A novel programmable query processing platform on fpga. *Microprocessors and Microsystems 51* (2017), 142–164.

[112] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.

[113] Scofield, T. C., Delmerico, J. A., Chaudhary, V., and Valente, G. Xtremedata dbx: an fpga-based data warehouse appliance. *Computing in Science & Engineering 12*, 4 (2010), 66–73.

[114] Skubalska-Rafajłowicz, E. Exploring the solution space of the euclidean traveling salesman problem using a kohonen som neural network. In *International Conference on Artificial Intelligence and Soft Computing* (2017), Springer, pp. 165–174.

[115] Smith, M. J. S. *Application-specific integrated circuits*, vol. 7. Addison-Wesley Reading, MA, 1997.

[116] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research 15*, 1 (2014), 1929–1958.

[117] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 553–564.

[118] Stützle, T., and Ruiz, R. Iterated local search. *Handbook of Heuristics* (2017), 1–27.

[119] Sukhwani, B., Min, H., Thoennes, M., Dube, P., Brezzo, B., Asaad, S., and Dillenberger, D. E. Database analytics: a reconfigurable-computing approach. *Micro, IEEE 34*, 1 (2014), 19–29.

[120] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (2014), pp. 3104–3112.

[121] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[122] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[123] Taha, H. A. *Operations research: An introduction (for VTU)*. Pearson Education India, 2005.

[124] Tang, X., Kislal, O., Kandemir, M., and Karakoy, M. Data movement aware computation partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), ACM, pp. 730–744.

[125] Terasic. Ata/sas hsmc card.

[126] Tieleman, T., and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning 4*, 2 (2012), 26–31.

[127] TPC-H. Tpc benchmark h standard specification revision 2.17.0.

[128] Tucker, G., Bhupatiraju, S., Gu, S., Turner, R. E., Ghahramani, Z., and Levine, S. The mirage of action-dependent baselines in reinforcement learning. *arXiv preprint arXiv:1802.10031* (2018).

[129] Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *AAAI* (2016), vol. 16, pp. 2094–2100.

[130] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), pp. 5998–6008.

[131] Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *Advances in Neural Information Processing Systems* (2015), pp. 2692–2700.

[132] Wang, H., Mansouri, A., and Créput, J.-C. Cellular matrix model for parallel combinatorial optimization algorithms in euclidean plane. *Applied Soft Computing 61* (2017), 642–660.

[133] Wang, J., Ersoy, O. K., He, M., and Wang, F. Multi-offspring genetic algorithm and its application to the traveling salesman problem. *Applied Soft Computing 43* (2016), 415–423.

[134] Warren, R. H. Small traveling salesman problems. *Journal of Advances in Applied Mathematics 2*, 2 (2017).

[135] White, T. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[136] Wilbaut, C., Hanafi, S., and Salhi, S. A survey of effective heuristics and their application to a variety of knapsack problems. *IMA Journal of Management Mathematics 19*, 3 (2008), 227–244.

[137] Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8*, 3-4 (1992), 229–256.

[138] Woods, L., and Eguro, K. Groundhog-a serial ata host bus adapter (hba) for fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on* (2012), IEEE, pp. 220–223.

[139] Woods, L., István, Z., and Alonso, G. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment 7*, 11 (2014), 963–974.

[140] Wu, L., Barker, R. J., Kim, M. A., and Ross, K. A. Hardware partitioning for big data analytics. *Micro, IEEE 34*, 3 (2014), 109–119.

[141] Wu, L., Lottarini, A., Paine, T. K., Kim, M. A., and Ross, K. A. Q100: The architecture and design of a database processing unit. *SIGARCH Comput. Archit. News 42*, 1 (Feb. 2014), 255–268.

[142] Wu, L., Lottarini, A., Paine, T. K., Kim, M. A., and Ross, K. A. Q100: the architecture and design of a database processing unit. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 255–268.

[143] Wu, M., Sun, Y., Gupta, S., and Cavallaro, J. R. Implementation of a high throughput soft mimo detector on gpu. *Journal of Signal Processing Systems 64*, 1 (2011), 123–136.

[144] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., and Krikun, M. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR abs/1609.08144* (2016).

[145] Xilinx. Vivado hls tools.

[146] Xu, R., and Wunsch, D. Survey of clustering algorithms. *IEEE Transactions on neural networks 16*, 3 (2005), 645–678.

[147] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. Apache spark: a unified engine for big data processing. *Communications of the ACM 59*, 11 (2016), 56–65.

[148] Zhang, X., and LeCun, Y. Text understanding from scratch. *arXiv preprint arXiv:1502.01710* (2015).

[149] Ziener, D., Bauer, F., Becher, A., Dennl, C., Meyer-Wegener, K., Schürfeld, U., Teich, J., Vogt, J.-S., and Weber, H. Fpga-based dynamically reconfigurable sql query processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 9*, 4 (2016), 25.