Managed Acceleration for In-Memory Database Analytic Workloads

The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–27

DRAFT

Impact of Managed Acceleration on In-Memory Database Analytic Workloads

Eoghan O'Neill[a*] , John McGlone[a] , Peter Kilpatrick[b] and Dimitrios Nikolopoulos[b]

[b]*HANA Cloud Computing, SAP*; [b]*Queen's University Belfast, Northern Ireland*

(*v3.6 released March 2011*)

In-Memory Databases, such as SAP HANA, enable new levels of database performance by removing the disk bottleneck and by compressing data in memory. The consequence of this improved performance means that reports and analytic queries can now be processed on demand.

Therefore, the goal is now to provide near real-time responses to compute and data intensive analytic queries. To facilitate this, much work has investigated the use of acceleration technologies within the database context. While current research into the application of these technologies has yielded positive results, they have tended to focus on single database tasks or on isolated single user requests.

This paper uses SHEPARD, a framework for managing accelerated tasks across shared heterogeneous resources, to introduce acceleration into an In-Memory Database. Results show how, using SHEPARD, multiple simultaneous user queries all receive speed-up by using a shared pool of accelerators. Results also show that offloading analytic tasks on to accelerators can have indirect benefits for other database workloads by reducing contention for CPU resources.

Keywords: OpenCL; heterogeneous computing; workload management; in-memory database; predictive analysis;

1. Introduction

The emergence of In-Memory Databases (IMDBs) has helped to erode the bottlenecks of disk access latency and thus speed-up query processing times. For analytic queries, having data in-memory avoids the need to read tables from disk and the data is always available for immediate processing. Since data analysis commonly focuses on a subset of table columns, this has led to many IMDBs employing columnar storage which has been shown to benefit analytic queries [1] by having column data stored contiguously in-memory and thus improving read performance due to reduced cache misses. This improvement in the performance of data analysis queries has led to the desire to move from fixed date reports or batch jobs to enable real-time analytics within IMDBs.

The consequence of this is that IMDBs now need to support a range of analytic queries from multiple users and provide results as fast as possible. One such IMDB, SAP HANA, supports custom queries that can be implemented and run within the database. Doing so means immediate access to the data, with no need to copy to an external application server, and also means that multiple analytic queries can be run simultaneously, allowing for better utilisation of system resources.

*Corresponding author. Email: eoghan.o-neill@qub.ac.uk

ISSN: 1744-5760 print/ISSN 1744-5779 online
© 2011 Taylor & Francis
DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

In a conventional system, these analytic workloads must share CPU resources, and therefore it is recommended that they limit the number of cores they use so as not to negatively impact other workloads.

One solution to improve performance of such workloads is to employ acceleration technologies such as GPUs, which have been shown to be effective [2][3][4][5].

Unfortunately, explicitly targeting specific database functionality to GPUs is inflexible, as it introduces a hardware dependency. In reality, deployed systems can vary in the set of processing resources they contain: some may have multiple GPUs, or include other acceleration technologies such as Intel Xeon Phi cards.

Prior work introduced SHEPARD [6][7], a framework for managing accelerated tasks. This framework decouples applications from device specific code by allowing hardware accelerated task implementations to be dynamically loaded and executed at run-time.

Using SHEPARD, analytic workloads from multiple users can be managed and load balanced across a set of shared heterogeneous processing resources. SHEPARD exists as an external management process, allowing it to manage requests from multiple users and applications and orchestrate the execution of tasks on a limited set of heterogeneous processing resources. This is necessary as a single application cannot have knowledge of other applications that may also wish to make use of shared accelerators.

This paper investigates the ability of SHEPARD to effectively manage analytic workloads from multiple users across a limited set of heterogeneous processors. Accelerated implementations of three common analytic algorithms, K-Means, K-Nearest Neighbours and Multiple Linear Regression are provided for experimentation.

Prior work investigated the application of SHEPARD to individual stand-alone applications, while this work integrates SHEPARD into an enterprise In-Memory Database. This required adding SHEPARD managed tasks to the database source code and verifying that SHEPARD can load and execute and add managed accelerated implementations without requiring the database to be reloaded.

Experiments focus on the real-world achievable performance improvements experienced by the user when all the overheads of the database are included and when multiple users must share a limited number of processing resources. This is in contrast to simply investigating raw performance speed-up of the algorithms when run in isolated scenarios on dedicated hardware.

Since these analytic queries are typically run alongside other database workloads, an OLAP [1] benchmark, SAP-H is used to simulate a base load on the database. By comparing the run-time of this benchmark when run alongside analytic queries that use the CPU only and SHEPARD managed implementations that can be offloaded to accelerators, results show how the database performance can be indirectly improved by reducing the number of tasks competing for CPU resources through the use of accelerators.

This paper makes the following contributions:

- **Managed Acceleration in an IMDB**
  Allow accelerators to be shared between all database users transparently and effectively.
- **Study of accelerated Queries in an IMDB**
  Experiments investigate the actual improvement observable by the user queries, resulting from accelerated implementations being transparently managed over a set of shared resources when compared to CPU implementations.

[1]OLAP - OnLine Analytic Processing

- **Effect of Offloading on IMDB Performance**
  Results also show how offloading accelerated workloads can indirectly benefit overall database performance by removing work from the host processor.

Section 2 discusses existing works that have aimed to introduce acceleration to databases. Section 3 outlines an in-memory database system used in the investigation and how it deals with custom analytic workloads currently. Section 3 also details how the SHEPARD framework has been introduced to enable and manage accelerated query execution on available processing resources. Section 4 discusses the experiments carried out, and presents results. Section 5 closes the paper with evaluation of the approach and future steps.

2. Related Work

The introduction of acceleration into databases has already been investigated and much of this work has focused on improving the performance of certain database operations over CPU implementations, or choosing whether a task should execute on the GPU or CPU.

Breß et al. studied the selection of accelerators for database queries [8–10]. In this work they focus on estimation of query execution time on the CPU versus a GPU, using statistical models. Using these estimates they demonstrate the ability to improve performance by dynamically allocating queries to the correct device by first estimating its performance. The queries they study, sort and scan, have clear cross-over points where the CPU or the GPU will perform better, depending on the input. The method proposed is able to self-tune and learn from successive performance observations allowing it to adjust at run-time to the devices it has available. This work shows that static allocation is not suitable for many workloads and clearly demonstrates the advantages of run-time costing to make better allocation decisions. However, there is no study of how the system copes with simultaneous users or existing workloads that are not controlled.

Rauhe et al. study parallel execution on CPU and GPU by using just-in-time (JIT) compilation of SQL queries to produce vectorised code for the CPU and use OpenCL for the GPU. Their results show that a single OpenCL code can run effectively on both GPU and CPU. Though CPU optimised code does outperform the OpenCL code, it does not do so to a large degree, leading the authors to argue that having a single code that can run on many devices outweighs the slight loss of performance. Using their map-reduce style, compute/accumulate pattern, they are able to run TPC-H queries on the GPU, outperforming the CPU in some cases [4].

These approaches assume a hardware dependency and lack the flexibility to adapt to other possible hardware configurations a database may be deployed to. Therefore, any system that wishes to introduce acceleration must be able to accommodate a multitude of available processing resources, otherwise there is wasted potential to accelerate workload if available resources are ignored.

Additionally, many of the investigations have considered isolated execution of workloads, seeking to optimise the performance of a single query by using either the CPU or an accelerator, or splitting execution over both resources.

Govindaraju et al. [2] implement a GPU sorting architecture on billion row data tables and are able to achieve significant performance improvement, even on modest GPU hardware. Bakkum et al. implement SQL SELECT queries on GPU. In the paper they show speed-up of up to 70x that of the CPU for certain data sizes and argue that SQL is a much more accessible way for database programmers to use GPUs, thus reducing the effort required to use such processors [3]. Krueger et al. investigate the use of GPUs in merge operations for table columns in in-memory databases. While the authors demonstrate speed-ups on certain parts of the merge process, they also caution that data transfer to accelerator devices can hinder achievable speed-up [5].

Sukhwani et al. [11] investigate the use of FPGA acceleration of queries within a database. In this work the authors not only demonstrate significant speed-up on analytic queries, but also a 94% reduction in CPU time as a result of offloading the queries to the FPGA, which frees the CPU to process other workloads. The promising results of this work clearly demonstrate that acceleration can offer benefits to the database ecosystem, beyond acceleration alone. This work used hard-coded FPGA implementations and all supported queries were passed to the FPGA. However, this work effectively motivates the benefits accelerators can bring to databases.

Fang et. al [12] investigate the use of GPU based compression schemes to minimise data transfer and improve processing performance for in-memory databases. Using the lightweight compression techniques in their paper, the authors are able to achieve an order of magnitude improvement in overall query execution time.

In these studies the assumption is that a user query has sole access to all system resources and does not account for the fact that many users may make simultaneous requests on the system. Therefore it is also necessary to accommodate the situation where multiple accelerated workloads wish to use shared processing resources.

While current works have motivated the promise of accelerating database workloads, showing improved performance and also the indirect benefit for other database loads by offloading work, they lack any mechanisms for managing multiple users and accommodating different platform configurations.

This paper seeks to look beyond simply the speed-up of certain workloads, but to also consider the full context of performance improvements when multiple user requests vie for shared resources.

3. Managed Analytic Queries on SAP HANA

This section outlines the In-Memory Database and how SHEPARD is used to manage accelerated analytic query implementations within the database.

3.1 SAP HANA

SAP HANA is an In-Memory Database and the manner in which it handles data differs from that of traditional disk-based databases. All data is stored in main memory, removing the high penalties of disk access times. Additionally, most tables can be stored in columnar format. This provides two major advantages over row based tables, the first being sequential and contiguous allocation of data in main memory for each column. For analytic queries where individual columns may be processed at a time, this contiguous storage can speed-up processing of the data due to reduced cache misses associated with random access and the ability to vectorise processing of this data. The second advantage comes in the form of table compression through dictionary encoding. As the data is column oriented, each unique data item in the column can be recorded in a dictionary and replaced by a single index reference to the position the data item resides in the dictionary. Using this scheme results in high levels of compression for columns with few unique values and is especially useful for strings [13].

3.2 Analytics on HANA via AFL

SAP HANA also provides the ability to extend the functionality of the database through its Application Function Library (AFL). The AFL allows custom code to be executed in a separate process within the database, negating the need to copy data across network to a separate application server. THE AFL fulfils the need for users to run complex tasks which cannot be described using SQL.

These added functions, often implemented in C++, can be called via a registered SQL procedure call to perform any custom task. When called by a user query, these functions execute immediately and thus will compete for shared system resources, and so the number of threads they use is often limited to avoid negatively impacting other workloads.

When executing an AFL implemented query a number of steps must occur, as documented in Figure 1.

AFL.i	- Copy Compressed Table Data
AFL.ii	- Pre-process/Decompress Data
AFL.iii	- Perform query task on data
AFL.iv	- Output results - commonly written to an output table

Figure 1.: AFL Process Steps

In step (AFL.i), HANA makes a copy of the compressed table data for the function to operate on, allowing transactions to continue on the main table. This step is controlled by the HANA database.

Steps (AFL.ii) - (AFL.iv) execute the custom developer code within a separate process.

In step (AFL.ii), the compressed data is decompressed to its native form, for example, floats or strings, in a process referred to as *"materialisation"*.

Once the data is in its native form the custom function code is then executed on the data to perform the desired tasks in step (AFL.iii). Typically, step 3 is implemented by the developer in C++ to consume the table data. The columnar nature of the data lends itself well to code vectorisation or acceleration. If a device such as a GPU is to be used, the developer must explicitly code for this. However, the developer cannot be guaranteed that other workloads are not also using the GPU, and so the responsibility will be on the developer to handle situations where the GPU may not have enough resources available to process their function.

Finally, in step (AFL.iv) the results of the function are output for the user to consume, typically by writing the results to another table.

A real-world use of the AFL is the Predictive Analysis Library (PAL) which enables a number of analytic queries, such as clustering and classification, to be performed on table data.

3.3 Managing AFL queries using SHEPARD

The availability of the AFL in HANA allows custom code to be run inside the database. Typically, the developer must implement the code necessary to perform the desired task. Unfortunately, for accelerators, the developer needs to add code to discover devices, and target execution as necessary. If proprietary tools, such as CUDA for NVIDIA GPUs, are used then if those supported devices are not present the developer will need to provide a fall-back implementation. Since AFL functions may execute concurrently within the database, as they can be called by different users, other tasks may already be executing on the GPU and the developer will not be able to assume sole device ownership, and thus must be concerned with device contention and lack of resources at run-time. Given that different deployments of an IMDB may be to diverse hardware platforms, with different device configurations, an effective way of enabling acceleration of AFL queries among multiple users is necessary.

To address these problems SHEPARD is introduced to the database. SHEPARD is a task management framework for accelerated implementations [6][7]. The SHEPARD framework is comprised of a number of components that are designed to decouple applications from fixed hardware platforms and allow them to execute tasks dynamically, depending on the hardware available on the platform they are deployed to.

The components of the SHEPARD framework are briefly described here:

- **Task Repository**
  The Task Repository stores device specific implementations for any functions that should be managed by the framework. These implementations can be in the form of pre-compiled OpenCL kernels or self-contained shared libraries that can be loaded and executed at run-time. Device specific implementations can be provided from a number of sources including device vendors, expert programmers and even third party libraries. This allows for a sensible separation of concerns with expert device developers concerned only with device specific implementation and application developers concerned with high-level tasks.

  Storing implementations separately therefore removes the need to include device specific code and device management in applications, instead delegating this to the Runtime component.

  Cost Models - Additionally, for each device specific implementation, cost models in the form of linear expressions are stored. These cost models are derived separately from prior observations of how the task performs on the target device. Therefore, if cost models are not already provided for an implementation, some benchmarking is required. These cost models are then made accessible to the SHEPARD runtime which uses them to estimate the expected execution time of a task at run-time.
- **SHEPARD Developer Library**
  The SHEPARD developer library allows high-level application developers to make use of accelerated implementations via special function calls, referred to here as "*Managed Tasks*". Managed tasks are simply special objects that take the name of the task the developer wishes to call and the parameters that are required. When these tasks execute at run-time, the library takes care of connecting to the SHEPARD runtime, requesting that the task is executed, and then loading and executing the implementation that the SHEPARD runtime provides. This means that the application developer does not need to include device specific code or manage devices explicitly. This is handled in the SHEPARD runtime. An example of a managed task call can be seen in listing 1.

Listing 1: Plug-in Task as called by a Developer

```
Shepard::ShepardTask task("DeltaMerge", "mergeDouble");

task.addParameter( _mainDict    , _mainDict->size()     );
task.addParameter( _mainIndex   , _mainIndex->size()    );
task.addParameter( _deltaDict   , _deltaDict->size()    );
task.addParameter( _deltaIndex  , _deltaIndex->size()   );
task.addParameter( _deltaInvMap , _deltaInvMap->size()  );
task.addParameter( _pValidDocIds, sizeof(_pValidDocIds) );

task.execute();
```

- **SHEPARD Runtime**
  The SHEPARD runtime orchestrates the execution of all Managed Tasks in the system. It is implemented as a stand-alone daemon process and can accept requests from multiple applications and processes. The SHEPARD framework represents each accelerator in a node as a task queue. This affords a number of benefits, including isolation of tasks on devices for security and prevents exhausting device resources that could result from multiple tasks pre-allocating memory, as well as estimating the execution time of the current work queued to each device.
  Having SHEPARD manage task placement allows devices to be shared among many processes. It also allows task performance to be automatically fed back to the repository which can then be used to improve cost models in the future.

The SHEPARD framework is summarised in Figure 2.

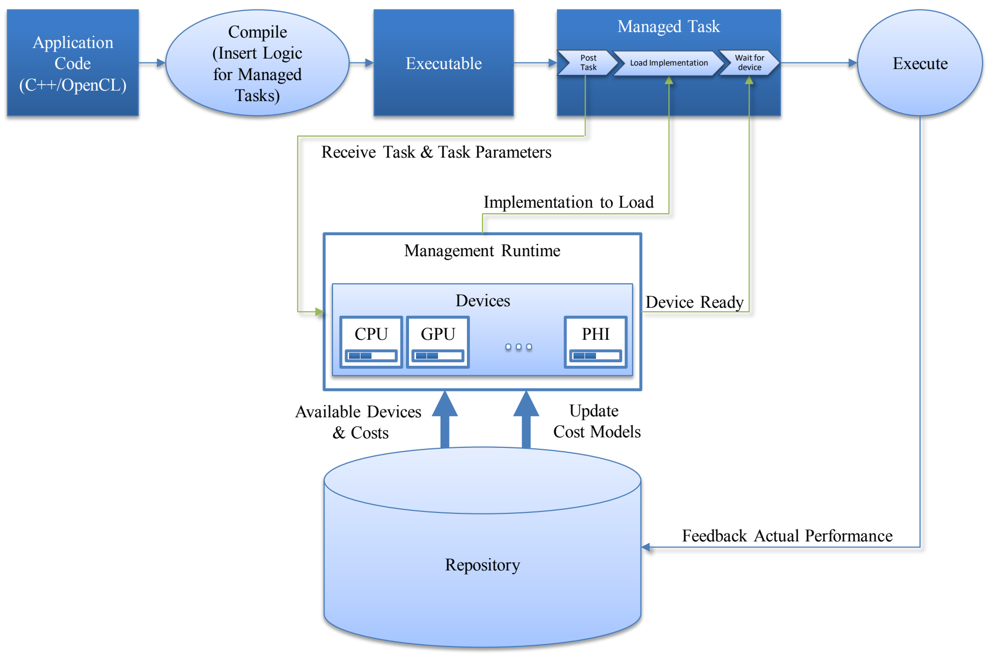


Figure 2.: The SHEPARD Framework.

3.4 SHEPARD Run-time Task Allocation

SHEPARD manages devices via a task queue based system and maintains a repository of implementations that can be loaded and executed at run-time. Therefore, instead of directly inserting code into the database via AFL, the developer registers their implementation with SHEPARD. The developer then simply needs to

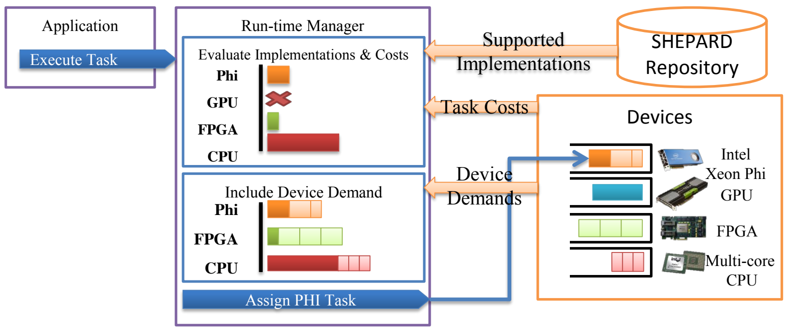


Figure 3.: The SHEPARD Runtime Allocation Scheme.

call the registered implementation via a SHEPARD task. SHEPARD takes care of choosing which implementations to load at run-time and when and on which devices to execute. This allows tasks to be updated in the SHEPARD repository without having to recompile or reload the database and allows AFL queries to be dynamically allocated to devices on any platform.

Each implementation registered in the repository also requires an associated cost model. This cost model is currently implemented as a linear expression, which is sufficient for the workloads studied in this paper, however the can easily be expanded to support other forms of cost model. The cost models for this work have be derived from observations of prior executions of each implementation on the target hardware. These cost models are then evaluated at runtime using the task parameters as inputs to the linear expression cost models. When evaluated the cost models yield the estimated execution time of a particular task implementation.

SHEPARD allocates tasks to devices as they are received. When a managed task is called, SHEPARD estimates its execution time for each supported device, using the cost model stored in the repository. Next, SHEPARD calculates the total execution time of tasks already queued to each device. Adding these two costs gives the total turnaround time for the task on each device. SHEPARD then places the task on the device which yields the lowest turnaround time. Tasks placed on a device queue are processed in order. This process is summarised in Figure 3.

Since the IMDB is a multi-user environment SHEPARD is required to accommodate simultaneous requests. By using the task queue mechanism, SHEPARD can estimate the current demand on each of its managed devices, based on the tasks in each device's queue, and by approximating the execution time of an incoming task, can determine which device can complete the task first. Using this mechanism, SHEPARD can effectively load balance tasks among devices and avoid any single device becoming a bottleneck.

SHEPARD does not attempt to prioritise queries or users; instead, it aims to place each task on the best available device as it is received.

Using SHEPARD to manage processing resources and implementations through the task queue mechanism allows multiple users to share access to these resources. This is important, as devices such as GPUs and Intel Xeon Phis have limited memory. If multiple users simultaneously attempted to load large datasets this could exhaust the memory resource. Therefore, accelerator management is a necessity for IMDBs where queries can be made at any time by any authorised user. As such, static allocation of workloads to resources is also not feasible and must be performed at run-time in order to adjust for when the system is under high or low demand. When the database is not under high-load it may be feasible to always send specific queries to the same device. However, under high-load, certain devices may receive high amounts of workload and, therefore, being able to dynamically allocate tasks is vital.

Further details on the SHEPARD framework can be found in [6, 7].

3.5 SHEPARD integration into AFL

With regard to AFL queries, SHEPARD is used to manage the execution of step (AFL.iii) using SHEPARD managed tasks. Step (AFL.i) is controlled by the database and therefore cannot be managed by the AFL developer and the materialization in step (AFL.ii) of execution is an AFL specific operation. Thus, with the data materialised, instead of writing device specific code to process it, a SHEPARD task is called to perform the analytic query. SHEPARD then determines which devices can execute the task, and of those devices, which can return a result the soonest. SHEPARD will then orchestrate when and where the task will run. This process is outlined in Figure 4.

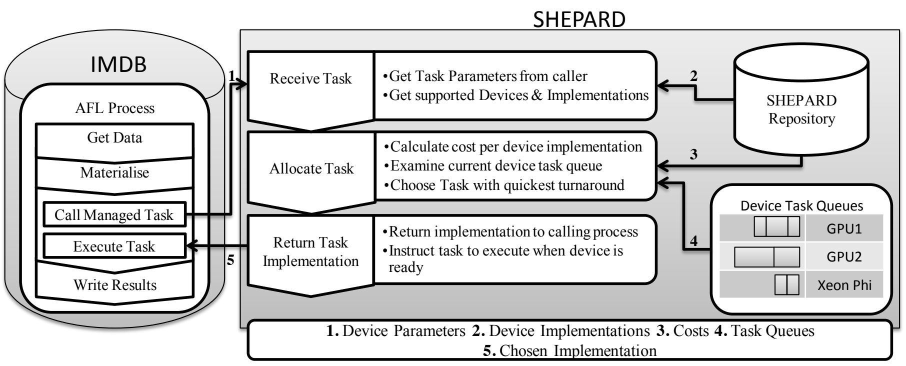

SHEPARD
IMDB
AFL Process
Get Data
Materialise
Call Managed Task
Execute Task
Write Results
1
Receive Task
•Get Task Parameters from caller
•Get supported Devices & Implementations
2
SHEPARD Repository
Allocate Task
•Calculate cost per device implementation
•Examine current device task queue
•Choose Task with quickest turnaround
3
Return Task Implementation
•Return implementation to calling process
•Instruct task to execute when device is ready
4
5
Device Task Queues
GPU1
GPU2
Xeon Phi
1. Device Parameters 2. Device Implementations 3. Costs 4. Task Queues
5. Chosen Implementation

Figure 4.: Managed analytic query process using SHEPARD and AFL.

4. Experiments

The experiments presented here investigate the application of SHEPARD managed tasks to analytic workloads within an IMDB. The experiments study the actual real-world performance of the full user queries within the database, as opposed to looking only at the accelerated portions of the queries.

In particular, experiments focus on scenarios where multiple user AFL queries compete for shared resources. This is in contrast to HPC workloads which may focus on large batch jobs or where jobs are optimised to use all available resources. In an IMDB scenario, resources must be shared between many users, and queries are made in an ad hoc fashion.

Results compare the accelerated SHEPARD managed tasks to those of unmanaged, multi-threaded CPU implementations, hereafter referred to as *"native"* implementations, and also examine how much of the total possible acceleration of each user query is actually realised.

The experiments are then repeated with the introduction of a base workload on the SAP HANA database, simulated using an OLAP benchmark, SAP-H. This time, results show how unmanaged CPU implementations and the SHEPARD managed accelerated implementations fare when the database is under load. In turn, the impact of the AFL queries on the base load is also examined to determine the indirect benefit to the database of offloading analytic workloads to accelerators.

4.1 Hardware Platform

The hardware platform consists of a dual-socket Ivy-bridge Xeon CPU with a total of 24 hyper-threaded hardware cores resulting in 48 logical cores being available. There are three accelerators cards present in the system, two Intel Xeon PHI 5110p cards and one NVIDIA K20x GPU. Further details are given in Table 1.

The mix of three architectures, CPU, GPU and Intel Xeon Phi, as well as the presence of two Xeon Phi cards demonstrates the potential of a wide range of compute resources being available in modern systems. Clearly, manually generating code to cover such a range of configurations is challenging for application developers.

Device	**Cores**	**Memory**
Intel Xeon x5650	24 (48 Threads)	256 GB
(2x) Intel 5110p	61 (240 Threads)	8 GB
NVIDIA K20x	2688	6 GB

Table 1.: Experimental Hardware Platform

4.2 Baseline - SAP-H

In order to quantify system performance, the SAP-H benchmark is used. SAP-H is an OLAP benchmark based on the TPC-H benchmark [14] that is used to generate a baseline for OLAP performance of a HANA system. As SAP HANA is an in-memory, primarily column based database, SAP-H uses tables and data that better reflect the reality of those found in a real SAP HANA system.

Using the SAP-H benchmark simulates a consistent baseline load on the IMDB, allowing degradation of system performance to be measured when additional workload is placed on the system. This base load also enables observation of how the analytic queries react when the system is already busy.

4.3 Studied Analytic Queries

The experiments reported here focus on three common analytic algorithms, K-Means, K-Nearest Neighbour and Multiple Linear Regression which cover the popular analytics categories, clustering, classification and regression, respectively. The choice of these algorithms is further motivated by their presence in the Predictive Analysis Library (PAL), which implements a number of common predictive algorithms used by businesses to extract information from data tables.

For each of the algorithms an OpenCL implementation is provided as well as a multi-threaded implementation obtained from the SAP HANA Predictive Analysis Library (PAL) which runs on the CPU [15].

Since each of the algorithms has been implemented using OpenCL, they can be targeted to GPU, CPU and Intel Xeon Phi technologies. The implementations are provided to SHEPARD as plug-ins, allowing SHEPARD to load and execute them on any available OpenCL devices.

K-Means

The K-Means algorithm performs cluster analysis on a dataset aiming to classify data points into one of K distinct classes. The algorithm works by first creating K class centres, which for this implementation are created by choosing the first K

data points. The algorithm then iteratively performs the following two steps until either a maximum number of iterations is reached, or no points change class.

(1) Assign each point to the nearest class centre,
(2) Update the position of each class centre according to its membership.

For the experiments carried out here, the K-Means algorithm operates a fixed number of 100 iterations.

Note that in the OpenCL implementation, the class update is performed sequentially on the CPU while the assignment stage is performed in SIMD fashion using OpenCL.

K-Nearest Neighbours (KNN)
The KNN algorithm seeks to classify data points according to a set of existing data that has already been labelled with class data, for example, by using K-Means.

To determine the class of a new data point, the KNN algorithm first identifies the nearest K data points to the new data point. Then the class labels of the K nearest neighbours are inspected to determine the most common class amongst the neighbours. The new point is then assiged to the most common class among its neighbours.

This process is repeated for every new data point that needs to be classified.

Multiple Linear Regression
Multiple linear regression is an algorithm that seeks to describe the relationship of an output variable Y (dependent variable) to a set of N input variables X (independent variables), as a linear equation, given a number of observations. This is depicted in Equation 1, where α refers to the intercept and β the coefficient value for the corresponding independent variable.

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_N X_N \tag{1}$$

This multiple linear regression algorithm takes a set of observations and derives the linear equation from these observations. The algorithm can be implemented using matrix multiplication to solve Equation 2

$$(A^T A)X = (A^T Y) \tag{2}$$

Given a set of N observations, A is an MxN matrix containing the values of the independent variables, and Y is a vector of size N containing the dependent variable values. X is a vector of size N which, when the equation is solved, will contain the coefficient for each independent variable.

4.4 Experimental Output

For each of the experiments, the performance of the system is examined from three view points.

- **Client Time**
  Time the user waits from the point of invoking the query until a result is returned. This covers steps (AFL.i)-(AFL.iv) of the AFL query process.
- **Total Server Time**
  This is the time taken to execute steps (AFL.ii)-(AFL.iv) of the AFL query,

i.e. the portion of the query that executes all the custom developer code in a separate process.

- **Algorithm Time**
  This concerns only step (AFL.iii) of the AFL query, i.e. the step that computes the K-Means, KNN or Linear Regression functions. It is this step that is controlled by SHEPARD to manage accelerated implementations.

4.5 Experiment Scenarios

The experiments cover a number of different scenarios which use various numbers of database users and dataset sizes. The purpose of these scenarios is summarised in this section.

- **10 Users** - This scenarios creates 10 database users that each execute a single instance of the same analytic query. This scenario uses the large datasets described in Table 2. This scenario is repeated 3 times, one for each algorithm, K-Means, KNN and Linear Regression. This scenario looks at an extreme case where multiple users are all executing long running compute intensive queries. This scenarios compares how the native and accelerated implementations perform for each query type.
- **5 Users** - This scenario creates 5 users that execute 20 analytic queries each. This scenario uses the smaller datasets outlined in Table 2. This scenario represents a light workload on the database. Limiting the number of users to 5 limits the resource contention in the system .
- **100 Users** - This scenario creates a large number of simultaneous users to each execute one of the analytic queries. In this scenarios there is a large demand place on the system with 100 users simultaneously attempting to execute either K-Means, KNN or Linear Regression. This scenario is intended to stress the system and examine how SHEPARD's queue based management scheme can actualise query acceleration on shared resources. Using 100 users creates a high degree of resource contention, which is avoided by using SHEPARD to manage access to accelerators.

For each experimental scenario, the SAP-H benchmark is also performed alongside the studied queries to observe how their performances degrade with additional workload. The SAP-H benchmark also provides a measure of how well the database is able to handle other queries in the system when compute intensive workloads are executing.

Note that in these experiments, the native queries are able to use all cores in the system to maximise their performance. In a real enterprise scenario, it is common to throttle the number of threads provided to these workloads to reduce the performance impact on other database workloads, at the cost of longer running queries. By introducing SHEPARD, these queries can run at full speed on accelerators since SHEPARD can load and schedule device specific implementations at run-time. Therefore, the native queries are allowed to run at full speed in these experiments for fair performance comparison.

	Large	**Small**
K-Means	100,000,000 x 2	10,000,000 x 2
KNN	100,000,000 x 2	10,000,000 x 2
Linear Regression	8,000,000 x 26	1,280,000 x 26

Table 2.: Algorithm Datasets (Rows x Columns)

4.6 10 Users - Large Datasets

In this experiment each user performs the same query using a large dataset. The datasets used contain multiple gigabytes of data resulting in these queries placing a sustained and significant load on the system. Therefore, this experiment looks at an extreme scenario where the database is required to service multiple very high load queries. To accommodate such a scenario, users would typically be expected to limit the number of threads that their queries use to enable fair use of resources, at the cost of much longer execution times. Using SHEPARD, however, these tasks can run on shared accelerators and for comparison, the native queries are executed using all threads to maximise their performance.

Results show that by using SHEPARD to share resources across multiple users, the accelerated queries can achieve significant speed-ups for all users when compared to the native implementations. Results also show that significant load is removed from the CPU by offloading these tasks, resulting in a reduced degradation to the SAP-H benchmark when SHEPARD is used.

These experiments use large datasets, shown in Table 2, to create significant load on the system. The result is that each query performed on these tables must process gigabytes of data, providing sustained demand on the system.

Three sets of experiments are performed on these datasets, one for each algorithm. Each experiment is performed with 10 users that simultaneously invoke either K-Means, KNN or Linear Regression. The experiments are then repeated with the introduction of the SAP-H benchmark which is run alongside the analytic workloads to observe the performance impact when a base load exists on the system.

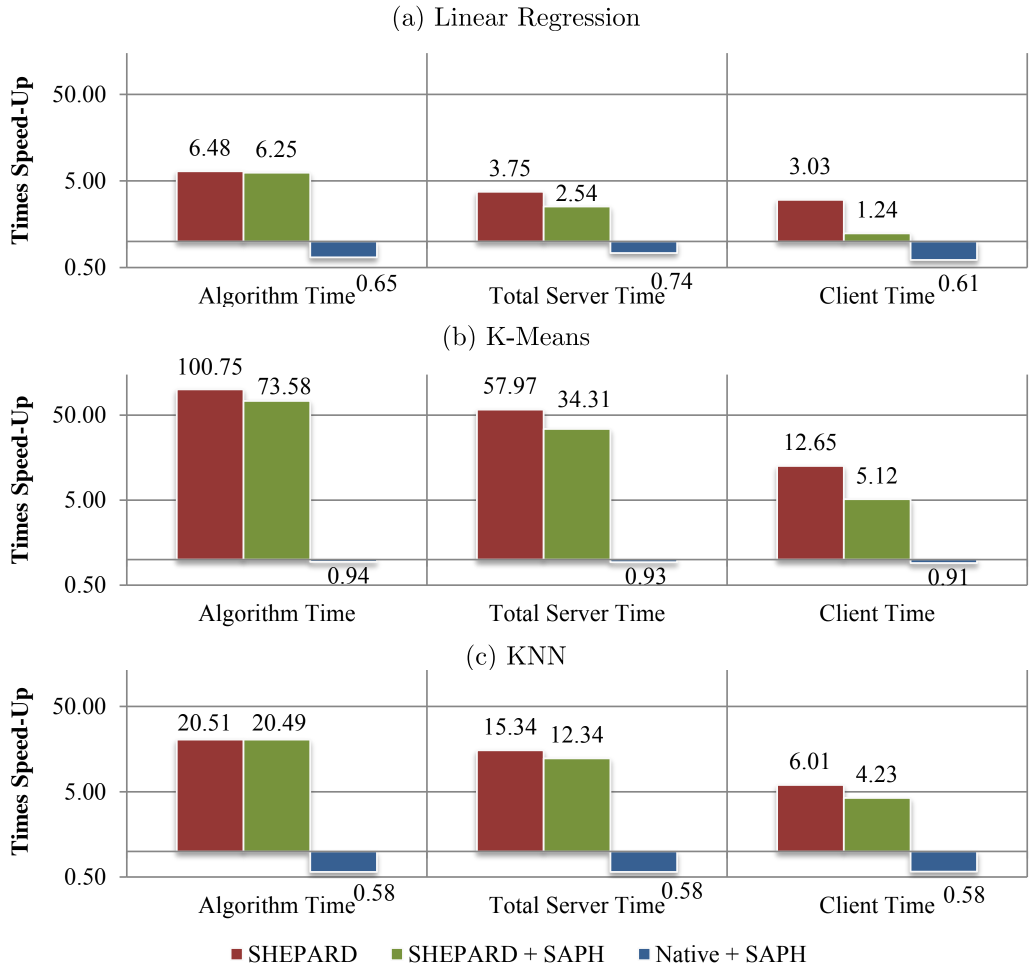


Figure 5.: 10 Users, Times Speed-up Versus Native Execution Time

Figure 5 shows the achieved speed-up from the accelerated implementations managed by SHEPARD, among the available devices. Notice that, since the K-Means implementation must transfer data to the host and update the class centres on the CPU, the K-Means algorithm suffers a significant performance penalty when the SAP-H benchmark is run as observed performance speed-up is reduced by over a quarter. The KNN and linear regression algorithms, on the other hand, are processed almost entirely on the device and not the host, resulting in minor performance penalties when SAP-H is co-run.

For the unmanaged CPU implementations, the existence of SAP-H results in a significant reduction in performance of the KNN and linear regression algorithms, which achieve only 65% and 58% of their performance compared to when SAP-H is absent. K-Means however, still achieves over 90% of its performance: this is likely due to the fact that the K-Means query takes significantly longer that the other queries, of the order of hours as opposed to minutes, and so, the additional execution time imposed by SAP-H is proportionally much less.

When the total server time is inspected, as shown in Figure 5, a similar trend is observed with respect to the performance penalty when SAP-H is present. Since the K-Means algorithm takes the longest time to process, a greater percentage of the total server execution time is spent on processing the algorithm, as opposed to materialising the dataset or writing results. As a result, any impact on the performance of the K-Means algorithm has a greater impact on the overall execution time.

When these results are viewed from the client's perspective, as shown in Figure 5, greater reductions in overall speed-up are observed for the SHEPARD managed implementations. The further reduction in speed-up is a result of the time taken to copy the data from the database to allow it to be processed. For such large data sets this creates a large performance penalty.

For linear regression, the overall observed speed-up for the client is reduced (from 3 times speed-up to just 1.24 times speed-up) when SAP-H is present. K-Means and KNN fare much better with significant speed-up still observed, even when SAP-H is present.

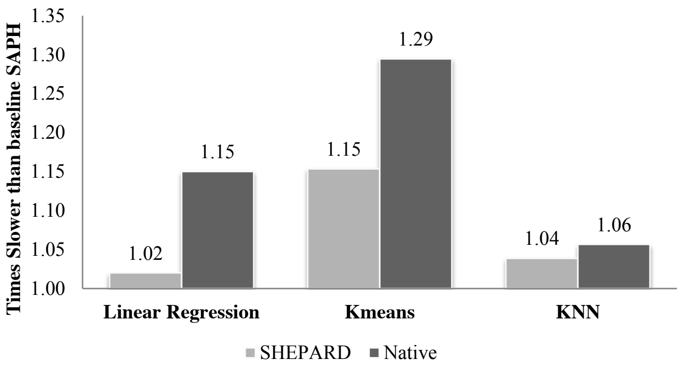


Figure 6.: 10 Users, SAP-H, Times Slower with Additional Workloads

Figure 6 compares the performance of SAP-H when run alongside the various workloads, to its baseline performance, i.e. when run with no other competing workloads. While linear regression achieved only a moderate improvement in query time, once data base overhead was taken into account, the benefit it provides to the host from being offloaded to accelerators, is significant. When the native implementation is used for linear regression, the SAP-H benchmark takes, on average, 15% longer. When SHEPARD is used to offload linear regression to accelerators, the performance penalty is reduced to just 2%, a significant saving. Offloading K-Means also achieves a significant saving, reducing the drop in SAP-H performance by half. However, due to the large dataset copying overheads and the host processing step in the K-Means implementation, even when the majority of K-Means is offloaded, there is still a 15% impact on SAP-H. Finally, for KNN, the impact is much less for both the native and managed executions, resulting in 6% and 4% SAP-H performance penalties, respectively.

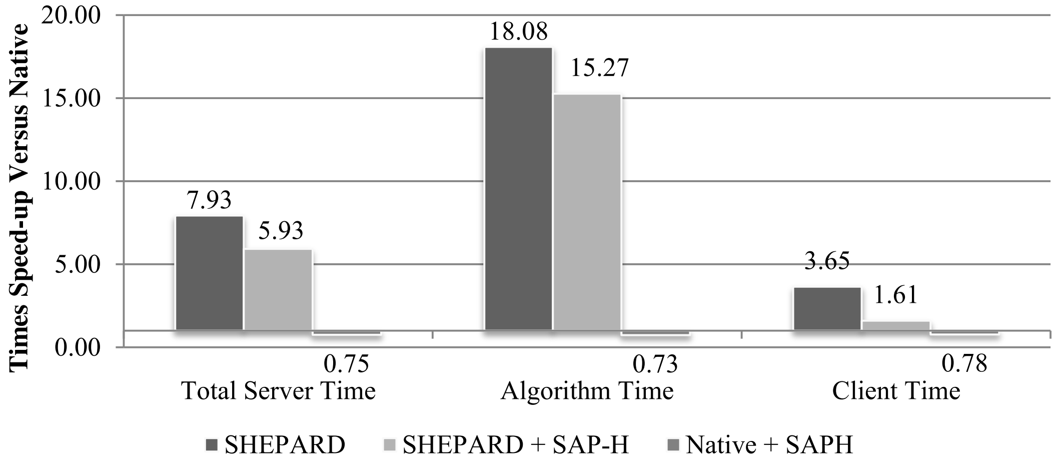


Figure 7.: 5 Users, Times Speed-up Versus Native Execution Time

4.7 5 Users - Mixed Queries

These experiments investigate behaviour when a small number of users each execute a set of queries, rather than just a single query. The results show that, for a small number of users, SHEPARD is able to adequately share processing resources among the different user requests without incurring significant delays. Results also show that, for this limited number of users and the reduced datasets, there is no discernible overhead on the SAP-H benchmark performance, as shown in Table 3.

# Users	SHEPARD	Native
5	1.00	0.98

Table 3.: SAP-H Benchmark Duration (5 Users) - Times Slower with additional analytic user workloads

While the previous experiments placed extreme load on the system from each user, these experiments focus on a smaller number of users (5), each with a set of 20 queries to perform. Each user performs a mix of K-Means, KNN and Linear Regression queries using the small datasets described in Table 2.

Figure 7 shows the average query speed-up observed over execution of the native implementations of the queries. As observed for 10 users, the accelerated algorithms achieve significant speed-ups over the native implementations. However, as with the 10 user experiments, the observed speed-up for the client is much less as the total execution of each query must also factor in copying the data and materialising the data.

However, even when compared against execution of the native implementations when SAP-H is not present, the SHEPARD managed execution with SAP-H is still able to achieve good speed-up, while native execution can only manage approximately 75% of the performance as compared to when SAP-H is not present

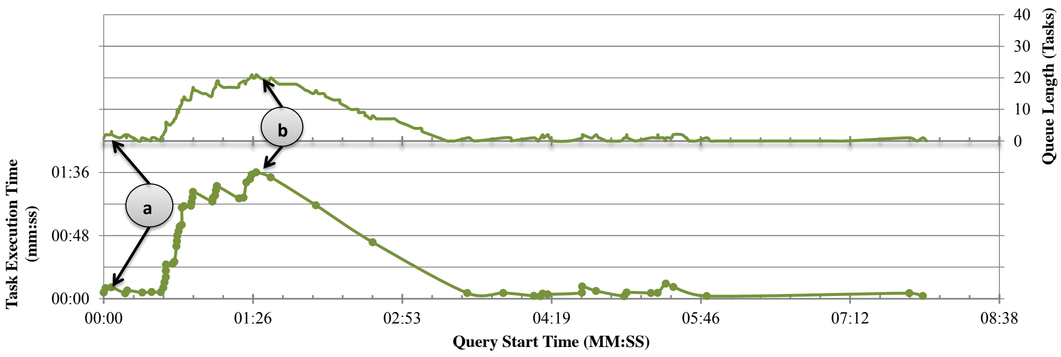


Figure 8.: Example Query Time-line Chart

Realised Acceleration Per Query

This section examines how much acceleration each user query receives. Since SHEPARD allocated tasks to devices dynamically it is important to determine that each device is effectively utilised, with an even distribution of tasks, such that no devices idle while other devices are over contended with too many tasks.

Additionally, since tasks are scheduling using device queues, there will be an overhead imposed on tasks in the form of how long they must wait on these queues before being processed. Also, depending on the existing demand on devices, a task may not be scheduled to execute on its preferred device. Given these constraints, this section determines how much acceleration each user query is still able to achieve when devices are shared.

Therefore to quantify how much of the potential acceleration each individual query receives, the observed execution time is plotted against the start time for each query. The results show both the execution time of a query in seconds, and its performance relative to the optimal execution time possible. The optimal execution time for the native queries occurs when they are executed on an idle system with access to all CPU cores. The optimal execution time for the SHEPARD managed accelerated queries occurs when they execute on their preferred device with a zero queue wait time.

To help the reader better interpret the results, Figure 8 presents an example of the charts presented later in the paper. In this figure, the x-axis shows the start time of a query. On the bottom half of the chart, the y-axis displays execution time. Therefore, each point on this graph shows when a query started, and how long it took to complete. The top half of the chart shows the total number of tasks queued for a particular device. Therefore, point (a) in Figure 8 shows the first query starting at time 00:00 and taking under 5 seconds to complete when there are 0 tasks on the device queue. Point (b) shows a task starting at 01:28 that takes 36 seconds to complete when there are already 20 tasks on the device queue. In the example there is a clear correlation between the number of tasks on a device queue and the execution time of tasks.

Figure 9 shows the breakdown of queries during the 5 users experiments.

Figures 9a and 9b show the absolute and relative execution time of the SHEPARD managed queries, respectively. Due to the fact that there are only 5 simultaneous users and 4 available devices on which to execute tasks, the device queues never exceed more than 5 tasks, and usually only have a single task in their queue. This means that there are no major spikes in query execution time resulting from the queuing of tasks. These figures also show that each device is utilised throughout the duration of the experiment, demonstrating that SHEPARD is able to load balance tasks and avoid any single device becoming a bottleneck and avoiding having devices idle unnecessarily.

It can also be seen that the GPU receives the most tasks: this is because, since SHEPARD can estimate task performance using the cost models stored in the repository, it can make intelligent decisions on which devices can process certain tasks faster. Therefore, the GPU being much faster at processing both KNN and Linear Regression tasks, will receive a higher quota of these tasks.

Figure 9b shows that for some queries, the relative performance can be as much as 5 times slower than an optimal query performance. The main reason for this is that the KNN and Linear Regression queries both have short execution times and perform best on the GPU. Therefore, any time spent waiting in a device queue or not being able to execute immediately on the GPU causes a higher proportional performance degradation to K-Means, which is longer running and less device sensitive. Therefore, when looking at the absolute query times in Figure 9a, there is no pronounced change in query execution times, with at worst some performance degradation in the order of seconds.

Compare this to Figures 10a and 10b which show the same experiments for the native multi-threaded CPU implementations. While the relative query execution time rarely exceeds 1.5 times that of an ideal query, notice that the execution times are in the order of minutes, as opposed to seconds for the SHEPARD managed accelerated queries.

Figures 9c and 9d show the results of the SHEPARD managed queries when SAP-H is present. Once again the device queues do not grow large and many queries achieve close to their optimal performance. This is due to that fact that when SAP-H is present the database has other workloads to service, and so it can take longer for the database to make the data available to each AFL query. As a result, notice that the spread of query start times is much greater; therefore each device has more time between queries and so is much more likely to finish a task before a new task is received.

In the presence of SAP-H, AFL queries do take on average longer to complete, with the longest query taking 20 seconds when no SAP-H is present, and just over 25 seconds when SAP-H is present, as shown in Figures 9a and 9c. When the native queries are examined, queries can take up to 4 minutes longer in the worst case, must worse that the performance degradation of the managed queries. This is simply due to the fact that when SAP-H is present these queries must contend for the same resource, namely the CPU. As a consequence, longer running queries can take significantly longer to complete. Thus, there is a clear advantage to offloading such queries to accelerators where they will be much less sensitive to other database workloads.

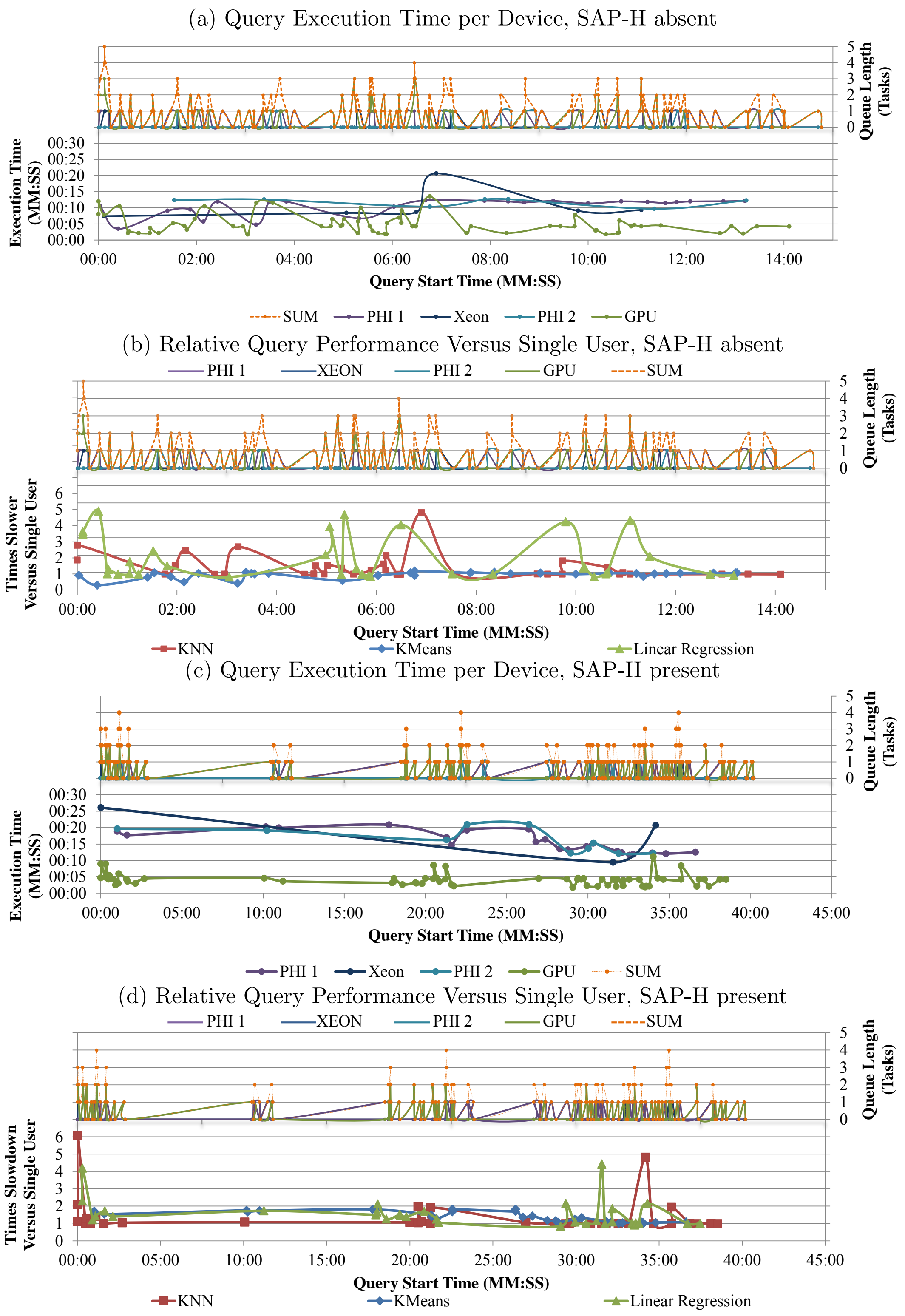


Figure 9.: 5 Users - Breakdown of SHEPARD Managed Query Performance
[top]Device Queue Lengths
[bottom]Query Execution Time (Algorithm Phase)

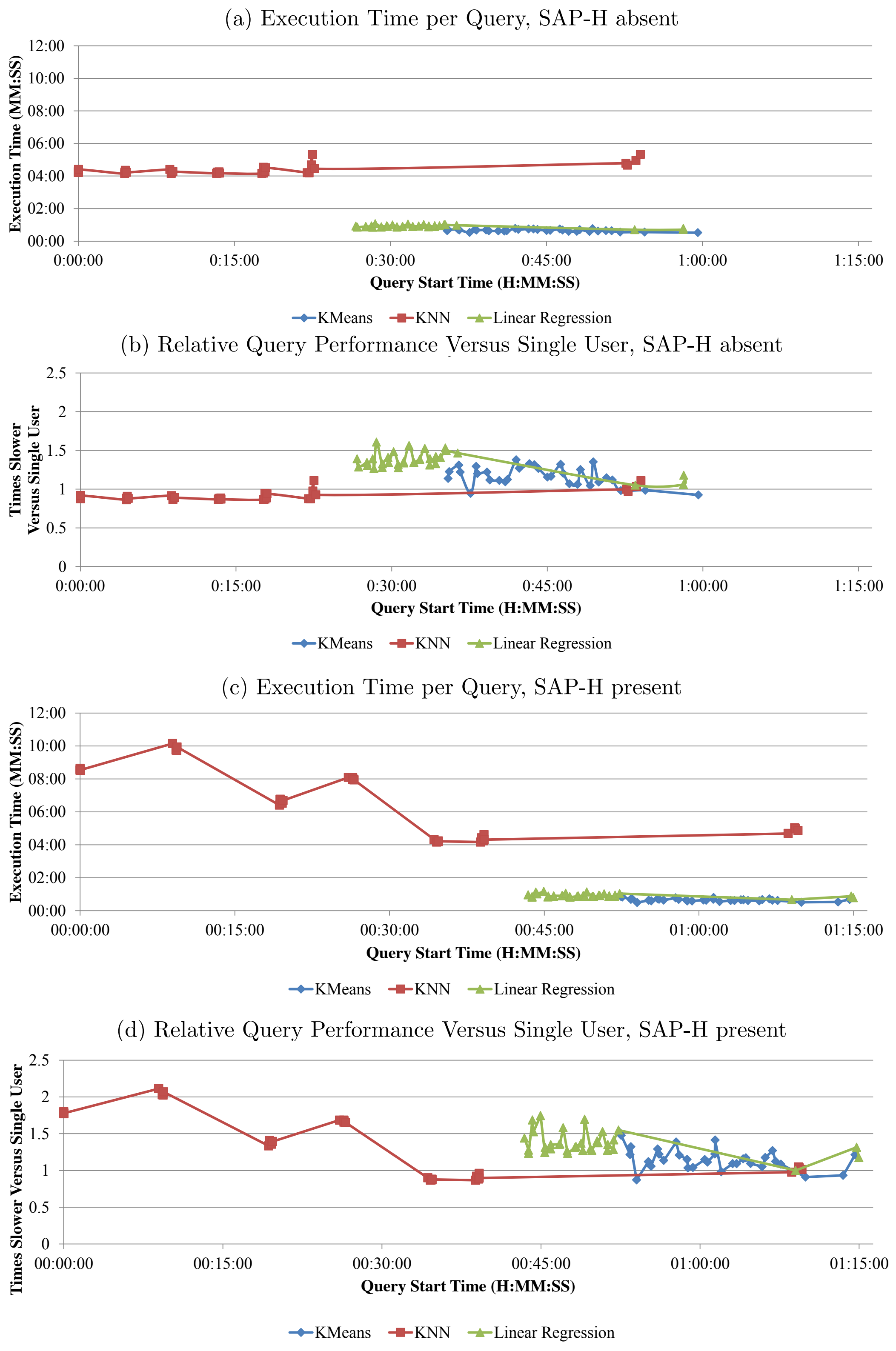


Figure 10.: 5 Users - Breakdown of Native Query Performance

4.8 100 Users - Mixed Queries

This scenario observes how performance is affected when a large number of users all wish to perform analytic tasks at the same time. In this scenario SHEPARD's task queues will grow large due to the limited number of resources shared among a high number of simultaneous requests. Despite this, SHEPARD still maintains good speed-up over the native implementations. Additionally, since SHEPARD allows all the tasks to be queued and mainly offloaded to the accelerators, degradation of the SAP-H benchmark performance is reduced from 14% to 8% when compared against the degradation suffered when SAP-H is run against the native implementations, as shown in Table 4.

In these experiments, 100 users each execute one query of either K-Means, KNN or Linear Regression and use the small datasets described in Table 2.

# Users	**SHEPARD**	**Native**
100	1.08	1.14

Table 4.: SAP-H Benchmark Duration (100 Users) - Times Slower with additional analytic user workloads

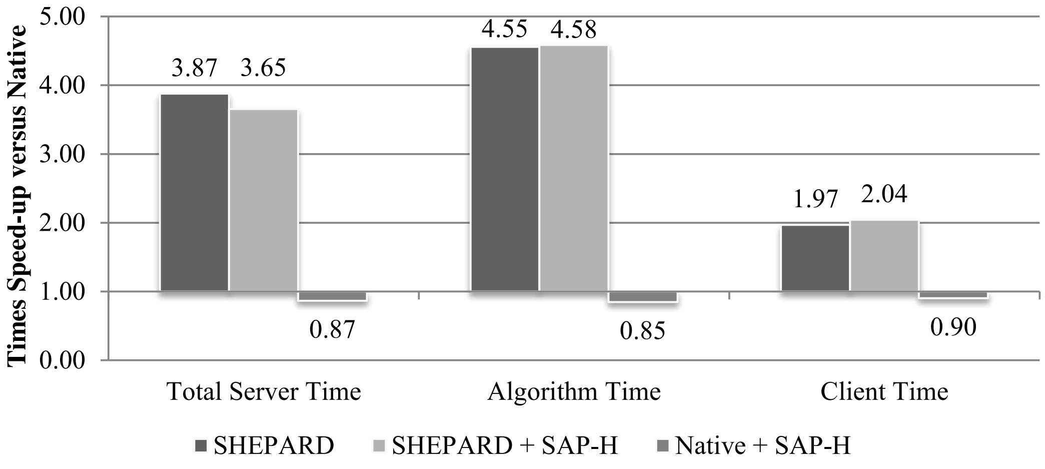


Figure 11.: 100 Users, Times Speed-up Versus Native Execution Time

Figure 11 shows the observed speed-ups of average query times over the native implementations. In this scenario, 100 users attempt to run one of 100 queries at the same time. For SHEPARD this results in many tasks being queued, as there are only four devices available to serve the requests. As a result, when SAP-H is co-run, there is little overhead seen on the algorithm time, i.e. the time taken for a managed task to execute, as time spent waiting in the queue is higher proportionally than any overhead of SAP-H. This carries forward to the overall speed-up observed on the client side as well, since allocating and executing tasks on the available devices now accounts for a much greater percentage of overall execution time for most queries.

For the native implementation queries, the introduction of SAP-H reduces the performance of the task as expected, though the slowdown observed at the client phase is less than that of the algorithm phase. This is due to the fact that 100 simultaneous users place significant strain on the system, and thus the data copy that occurs before the server can begin processing the task has much more influence on the overall execution time.

Realised Acceleration Per Query

When examining the performance of individual queries with 100 simultaneous users, much more pronounced performance impacts can be observed.

For the SHEPARD managed queries there is a clear peak in demand on the system which culminates in a large number of queued tasks for each device in the system. This correlates to increased query execution times for each query, as shown in all charts in Figure 12. Notice, however, that the peak number of tasks for each device is not the same, due to the fact that SHEPARD can account for the expected execution time of each task and also, importantly, account for the current execution time of each device queue. Observe that in Figure 12a, the GPU receives significantly more tasks than the other devices, yet peak query execution time per device is almost identical. This shows that SHEPARD can successfully load-balance tasks among all devices and no single device becomes a bottleneck and no queries spend inordinately long periods of time on any single device.

When the relative managed query performance is observed in Figure 12b, there is a clear difference between the query types. K-Means is both the longest running query and the least device sensitive, meaning that it performs similarly on each device. Therefore, its relative performance penalty is much smaller than the other queries. Conversely, Linear Regression is the shortest running query and performs 4 times better on the GPU than the next best device, therefore, resulting in a large performance degradation during peak load times. Thus given that the queue length peaks at over 01:30 during highest load, short running tasks will show a significant performance penalty. Therefore, future work should seek to further optimise SHEPARD's queuing strategy to account for such latency sensitive workloads, such as Linear Regression, especially in the presence of insensitive workloads such as K-Means.

When SAP-H is introduced in Figures 12c and 12d a performance penalty of at most 1 minute is observed during the peak load phase of the experiment. This results is a more prolonged peak in execution time; however, similar performance trends to when SAP-H is absent are observed in regard to the number of tasks allocated to each device and the fact that no device becomes a bottleneck. Thus, it can be seen, that even in the presence of contention, as introduced by SAP-H, SHEPARD's task allocation strategy is able to maintain comparable queue loads for each device resulting in a balanced workload with each device receiving an appropriate allocation of tasks according to the capabilities of that device.

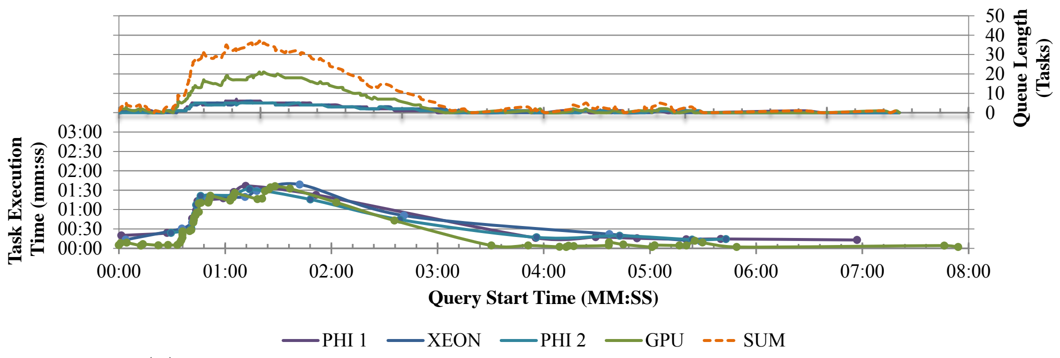
Figure 13 shows the same experiment for the unmanaged native query implementations. In this case, the relative performance as shown in Figures 13b and 13d is much more pronounced than the 5 user case, resulting in queries that are up to 12 times slower than the best case. Similarly to the managed queries, longer running queries suffer less in relative terms.

Figures 13a and 13c show the query times for each native query when SAP-H is absent and present, respectively. From these figures it can be seen that many queries suffer performance degradation when other database workloads are present. While SHEPARD is able to offload tasks to accelerators, the unmanaged tasks must compete for CPU resources with each other and with the SAP-H queries. This results in native AFL queries taking over 3 minutes longer in the worst case, as can be seen for the KNN query.

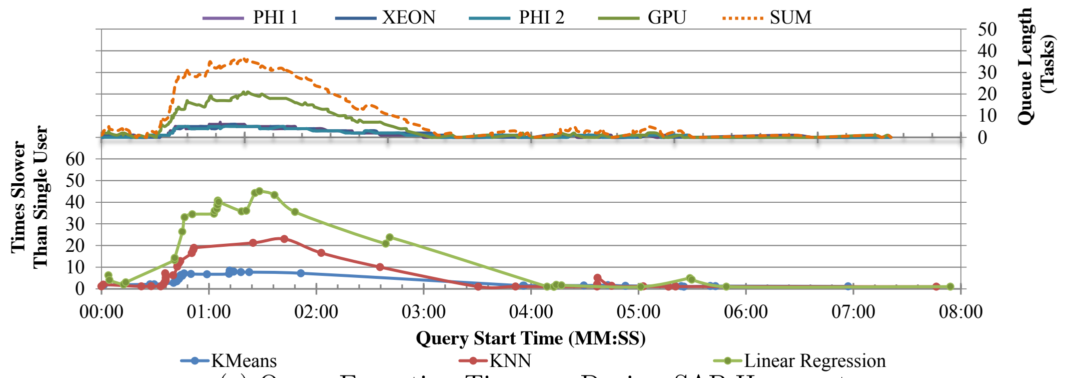
The results show that when large numbers of simultaneous users are present significant performance penalties can occur. For the native queries, without management, this is undefined behaviour and the user will have no idea of when their query will return as the system places no management on it. Additionally, unmanaged workloads will compete with other database workloads for CPU resource, harming overall system performance.

For the SHEPARD managed queries, the tasks are effectively load-balanced such that each device has an appropriate share of tasks according to how each task can perform on that device and the current demand placed on each device. The result is that at peak times, whether SAP-H is present of not, the maximum wait time imposed by each device queue is similar, resulting in a good load balancing outcome. This means that each device processes all the tasks they are given in a similar time, resulting in all devices being constantly occupied with work and no single device becoming a bottleneck in the system. Additionally, since SHEPARD estimates the execution time of a task and the queue wait time, this information can be quantified and passed back to the user if necessary, which is not possible for unmanaged queries. Using SHEPARD to manage tasks and place them on accelerators also has the added benefit of reducing CPU resource contention and therefore has much less impact on other database workloads, as evidenced via SAP-H.

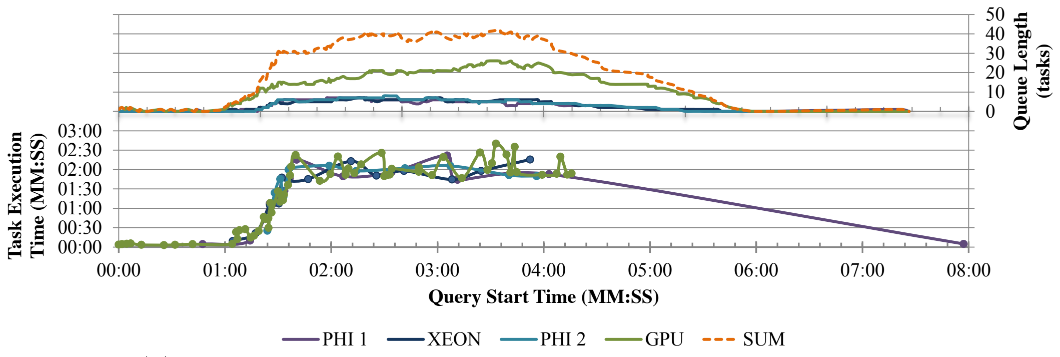
(a) Query Execution Time per Device, SAP-H absent



(b) Relative Query Performance Versus Single User, SAP-H absent



(c) Query Execution Time per Device, SAP-H present



(d) Relative Query Performance Versus Single User, SAP-H present

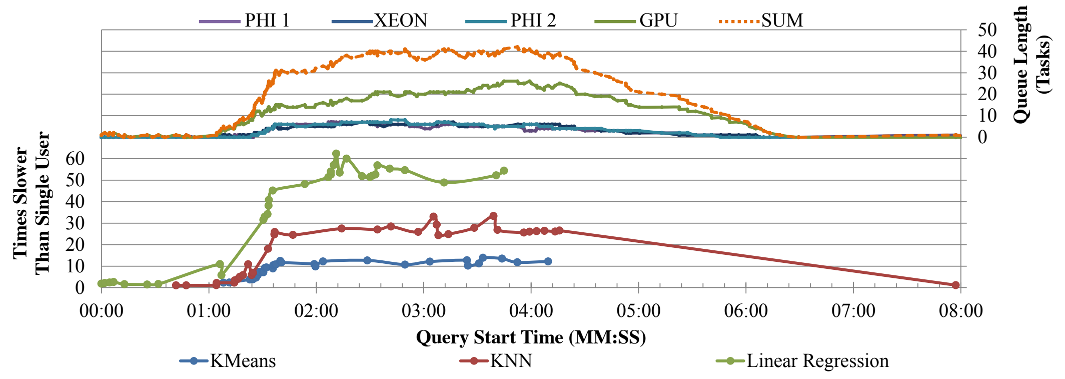


Figure 12.: 100 Users - Breakdown of SHEPARD Query Performance
[top]Device Queue Lengths
[bottom]Query Execution Time (Algorithm Phase)

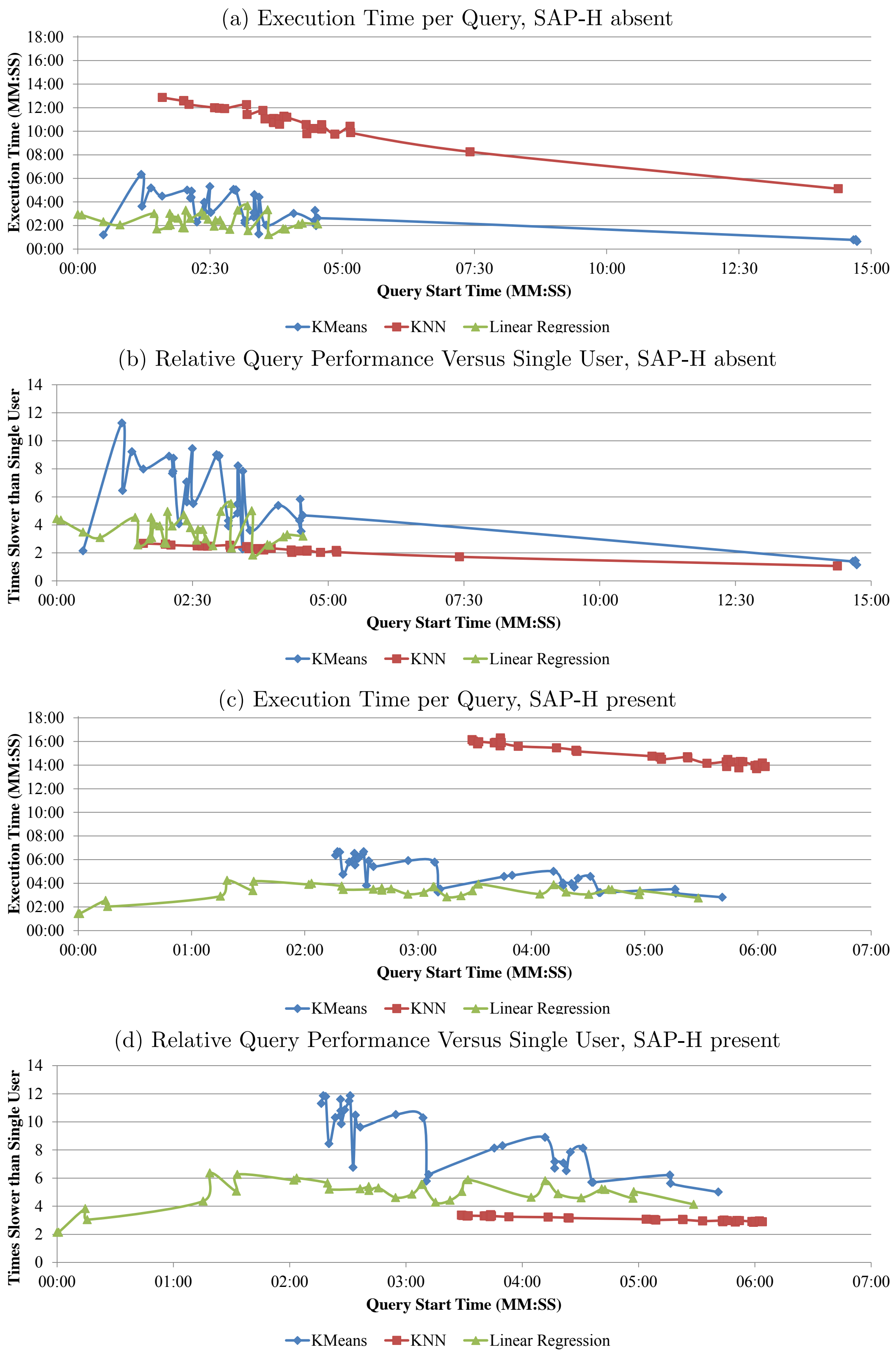


Figure 13.: 100 Users - Breakdown of Native Query Performance

5. Conclusions & Future Work

This work has presented an investigation into managing accelerators within an IMDB to speed up a set of analytic queries with various numbers of users. Using SHEPARD, a limited set of processing resources is shared among up to 100 users, all wishing to execute accelerated implementations.

Results show that by using SHEPARD to manage accelerated tasks, average query time for all users can be reduced.

The context of executing within the IMDB environment has repercussions on the observed performance by the client calling the queries. The database imposes an overhead of copying data and pre-processing it before it can be used. This reduces the overall performance improvement and is susceptible to performance degradation when co-run with the SAP-H benchmark.

For the database performance itself, simulated using SAP-H, results show that using SHEPARD to offload much of the additional query workload to accelerators can reduce demand on the CPU and improve general database performance.

In the future this work can be extended to dynamically adjust the costs of accelerated workloads in SHEPARD. Although static costs did not result in poor allocations in the experiments presented here, it could do so on other workloads and is an issue worth investigating. While in this work each user and query was treated equally, additional database oriented allocation strategies could be implemented that would allow SHEPARD to prioritise certain users or workloads.

References

[1] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM, 2008, pp. 967–980.

[2] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data.* ACM, 2006, pp. 325–336.

[3] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA : Extended Results University of Virginia Department of Computer Science Technical Report CS-2010-08," *Science*, 2010.

[4] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber, "Multi-level parallel query execution framework for cpu and gpu," in *Advances in Databases and Information Systems.* Springer, 2013, pp. 330–343.

[5] J. Krueger, M. Grund, I. Jaeckel, A. Zeier, and H. Plattner, "Applicability of gpu computing for efficient merge in in-memory databases." in *ADMS@ VLDB*, 2011, pp. 19–26.

[6] E. O'Neill, J. McGlone, J. Coutinho, A. Doole, C. Ragusa, O. Pell, and P. Sanders, "Cross resource optimisation of database functionality across heterogeneous processors," in *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, Aug 2014, pp. 150–157.

[7] E. Oneill, J. McGlone, P. Milligan, and P. Kilpatrick, "SHEPARD: Scheduling on heterogeneous platforms using application resource demands," in *Proceedings - 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014*, 2014, pp. 213–217.

[8] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake, "Automatic selection of processing units for coprocessing in databases," in *Advances in Databases and Information Systems.* Springer, 2012, pp. 57–70.

[9] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake, “Efficient co-processor utilization in database query processing,” *Information Systems*, vol. 38, no. 8, pp. 1084–1096, 2013.

[10] S. Breß, I. Geist, E. Schallehn, M. Mory, and G. Saake, “A framework for cost based optimization of hybrid cpu/gpu query plans in database systems,” *Control and Cybernetics*, vol. 41, 2012.

[11] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, “Database analytics acceleration using fpgas,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 411–420.

[12] W. Fang, B. He, and Q. Luo, “Database compression on graphics processors,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 670–680, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1920927http://dl.acm.org/citation.cfm?id=1920927$\backslash$nhttp://portal.acm.org/citation.cfm?id=1920927

[13] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, “The sap hana database–an architecture overview.” *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[14] “TPC-H Benchmark, Transaction Processing Performance Council,” http://www.tpc.org/tpch/, [Online; accessed June-2015].

[15] SAP, “SAP HANA Predictive Analysis Library (PAL),” help.sap.com/hana/SAP_HANA_Predictive_Analysis_Library_PAL_en.pdf, [Online; accessed June-2015].