Knowledge Accelerated Algorithms and the Knowledge Cache

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree
Master of Science in the Graduate School of The Ohio State
University

By

Matthew Goyder, B.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2012

Master’s Examination Committee:

Dr. Srinivasan Parthasarathy, Advisor

Dr. Gagan Agrawal
© Copyright by
Matthew Goyder
2012
Abstract

Knowledge discovery through data mining is the process of automatically extracting actionable information from data, that is, the information or knowledge found within data which provides insight beyond that which may be found by observing the cardinal state of the data itself. This process is human driven; there is always a human at the core.

Knowledge discovery is inherently iterative, a human discovers information by posing questions to a data mining system, which in turn provides answers. New questions are developed upon receipt of these answers and these new questions are asked. Clearly these answers need to be provided in as timely a fashion as possible in order for the human at the core to form ideas and solidify hypotheses. Unfortunately many questions take too long to be answered to be useful to the human. Is there anything we can do to speed up the response to these questions if the answer is based in part upon answers previously provided?

What we can do is when a query (question) is submitted (asked) to a data mining system, we can store the result (answer) as well as information about the result in a cache and then re-use this information to help respond to the next query in a more timely fashion. If a query partially contains a result which was found in the past, we can combine this information with new information to provide the result much faster than if we were to re-run a query incorporating no prior information.
This thesis explores this idea by introducing a high performance information cache called a Knowledge Cache with remote access capabilities, as well as a programming model and API for clients to both store, query, share and retrieve knowledge objects from within it. These knowledge objects can then be used in conjunction with a modified data mining algorithm to reduce query processing time for new queries where prior information is useful. We explain the usage model of the Knowledge Cache and API, as well as demonstrate performance gains by using the Knowledge Cache in the context of two classic data mining algorithms: $k$-means clustering and frequent itemset mining.
For Charlotte Gray Goyder. One day, I hope you find this, and smile.
Acknowledgments

I would like to extend my deepest thanks to my graduate academic committee Gagan Agrawal and my advisor Srinivasan Parthasarathy. Gagan has offered me support and leadership during my graduate career. I would like to extend special thanks to Srini for his patience and insight, as well as for his student advocation. Srini’s passion and drive will cultivate a long lasting legacy of excellence.

I would like to extend thanks to the members of the Data Mining Research Lab for making my time at Ohio State special, namely Sitaram, Shirish, Greg, Venu, Xintian and Duygu. I would also like to thank the members of the Computer Science program office for their continued support and assistance.

I would also like to thank my family, my!wife Leighanne and my daughter Charlotte for their unwavering support. Leighanne has walked with me through periods of uncertainty and doubt and has always steered me onto the right path. Charlotte provides me with endless joy and I am privileged to be able to guide her as she grows. My family are my raison d’être.

This material is based upon work supported by the National Science Foundation under Grant No. #NGS-CNS-0406386 and #RI-CNS-0403342. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 The Knowledge Cache and API</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Thesis Statement</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Related Work</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Organization of this Thesis</td>
<td>6</td>
</tr>
<tr>
<td>2. Knowledge Caching</td>
<td>8</td>
</tr>
<tr>
<td>2.1 The Knowledge Cache</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 Design Goals</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 High Level Design Overview</td>
<td>11</td>
</tr>
<tr>
<td>2.2 The Knowledge Caching API</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Knowledge Cache Logic Flow</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Store Knowledge</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Retrieve Knowledge</td>
<td>20</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 System overhead of a Random query workload.</td>
<td>58</td>
</tr>
</tbody>
</table>

viii
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Knowledge Cache High Level Organization</td>
<td>13</td>
</tr>
<tr>
<td>2.2 UML Representation of the Knowledge Caching API</td>
<td>19</td>
</tr>
<tr>
<td>3.1 A Trie Encoding of Frequent Itemsets and the Linearized Form</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Processing time (in seconds) for $k$-Means clustering workloads, single client, varying server nodes.</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Random workload.</td>
<td>51</td>
</tr>
<tr>
<td>4.3 Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Structured30 workload.</td>
<td>52</td>
</tr>
<tr>
<td>4.4 Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Structured60 workload.</td>
<td>53</td>
</tr>
<tr>
<td>4.5 Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Structured90 workload.</td>
<td>54</td>
</tr>
<tr>
<td>4.6 Processing time (in seconds) for frequent itemset mining workloads, single client, varying server nodes.</td>
<td>54</td>
</tr>
<tr>
<td>4.7 Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Random workload.</td>
<td>56</td>
</tr>
<tr>
<td>4.8 Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Structured30 workload.</td>
<td>57</td>
</tr>
</tbody>
</table>
4.9 Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Structured60 workload. . . 57

4.10 Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Structured90 workload. . . 58
Chapter 1: INTRODUCTION

1.1 Introduction

As humanity thunders along in the information age we find ourselves producing and storing more information than ever before. More devices meet the hands of users daily, from the ultra low-cost laptops in rural Africa to the desktop-powerful tablets we carry to the smart-phones that live in our pockets, and as availability increases the sheer volume of data that is generated explodes. As more people become connected, more data is generated around their online existences and can include browsing habits, purchasing trends and media content consumption. Year on year the amount of information produced and replicated globally doubles with 2011 peaking at 1.8 Zettabytes [5]. The surge of data growth is not limited to online users; modern business institutions store a record of each transaction they make, be it a clothing purchase in a department store, trades made from an individual’s investment account or a package being sent across the world. Data is the key to a competitive edge in the modern world and companies are not secretive about the fact that they want it - and they are willing to offer discounts to customers in return for willing cooperating in providing them the information they seek. We carry shopping cards which are scanned at each purchase, and explicit language regarding the storage and analysis of
our data is contained within most end-customer agreements which permits this data to be collected, analyzed and identified to you. This data contains within it a treasure trove of knowledge enabling analysts to explain reasons behind shopping trends and give management evidence to make strong predictions about business outlook. In the scientific community huge quantities of data is collected through experimentation (LHC [19]), crowd-sourced projects (SETI Live-x [27], Folding@home [7]), computer simulation of natural phenomenon and medical imaging. Computer and storage infrastructure has increased in its ability such that storing this data is effortless, but with this ever increasing mountain of information comes a parallel challenge of how to retrieve and analyze this data quickly to access the secrets within.

Knowledge discovery through data mining is the process of automatically extracting actionable information from data, that is, the information or knowledge found within data which provides insight beyond that which may be found by observing the cardinal state of the data itself. This information can be used to make scientific discoveries, model phenomenon as well as give private enterprise a business advantage. This process is human driven; there is always a human at the core. A human poses questions to a data mining system and in turn receives an answer. Knowledge discovery through data mining on huge amounts of data is difficult and time consuming. Useful data mining algorithms can scale exponentially with the quantity and dimensionality of data, consuming huge amounts of processor resources. Knowledge discovery is also iterative, a human discovers information by posing questions to a data mining system, which in turn provides answers. New questions are developed upon receipt of these answers and new questions are asked. Clearly these answers need to be provided in as timely a fashion as possible in order for the human at the
core to form ideas, solidify hypotheses and discover parameters which provide the best answers.

1.2 Problem Motivation

During the process of knowledge discovery data mining algorithms will be re-executed as the user is exploring the data space and finding the information they need. During these repeated executions data access patterns may be repeated, along with re-computation of results discarded from the previous execution [20, 10, 8, 17, 23, 30]. This repetition is redundant and a waste of time, computing resources and pose a disadvantage to the user by contributing to slower discovery of actionable information. Can we identify this repeated computation and re-accessed data within the execution of an algorithm? What if this information was ready prior to execution and could be used to accelerate the execution of a subsequent algorithm? If this information can be characterized, extracted and stored could it possibly be retrieved and used by integrating it into an algorithm then boosting its performance across successive iterations?

1.2.1 The Knowledge Cache and API

To address these broad questions we present the Knowledge Cache. The Knowledge Cache is a facility through which a user may programmatically access a cache of knowledge objects which is a store of knowledge extracted from previous executions of algorithms which have been bundled and cached for later access. The Knowledge Cache is a high-performance service which services user-generated queries and returns user-defined sections of memory which has meaning to the user. The user decodes this information and uses it to accelerate data mining algorithm performance.
Knowledge Cache runs as a service across a user-specific number of cluster nodes, and uses the aggregate node memory capacity to store, index and retrieve knowledge objects. To provide programmatic access to the Knowledge Cache, we also present the Knowledge Caching API which gives a user a set of simple intuitive classes and templates to build, store, index, rank and transmit knowledge objects to and from the Knowledge Cache.

1.3 Thesis Statement

Exploratory data mining often results in the redundant repetition of previously performed work. We can characterize and extract the work an algorithm does and re-introduce it into a different execution of that algorithm to realize an increase in performance. A generic system for storing and retrieving this information is key in accelerating algorithms in this fashion.

1.4 Related Work

Nag et al. [23] introduce the idea of using a knowledge cache for storing itemset support counts to avoid repeated database scans by a frequent itemset mining algorithm. They store the support of itemsets generated in previous queries in various cache configurations, and explore various cache replacement schemes which include FIFO replacement and more advanced tree based techniques which afford better pruning and itemset retrieval. The authors use a single dataset and their query workload is made up of a series of whole-dataset queries with varying support levels. Their work differs from this thesis in several key aspects: it is limited to frequent itemset mining only, they use a small fixed-sized cache of 30MB or less, support a single
client only, and in the context of frequent itemset mining, they do not support range queries. The Knowledge Cache presented in this thesis is algorithm-independent and can store any data that a client packs into a knowledge object, this includes knowledge objects built from multiple datasets and from queries with varying metadata. The system takes advantage of the aggregate memory of the nodes which compose the server supporting many gigabytes of data storage. The Knowledge Cache can service queries from multiple connected clients and the knowledge stored by each client can be delivered to any other client, if it’s metadata is a close match for a new query. Our example implementation for frequent itemset mining is based on an incremental update algorithm which can receive a knowledge object with higher support and a different data range and still use it to quickly discover emerged itemsets.

Nag et al. again use caching in [24] at a systems level for optimizing multidimensional data mining queries. This work is targeted at a OLAP audience and deals directly with satisfying frequent itemset mining queries expressed in SQL. OLAP makes heavy use of the ‘group-by’ clause and the authors seek to optimize processing these queries by caching various parts of the query space for faster retrieval when the query is broken down into its constituent parts. This thesis does not decompose queries but rather treats them as an atomic entity, and does not operate in such a focused application domain.

Parthasarathy et al. [25] is among one of the earliest to look at the notion of caching for data mining. The InterAct system is a data sharing mechanism in a client/server environment which manages data sharing across disparate processes. The idea being that a user can have a local cache of a dataset or data structures, and enjoy faster algorithm completion time by utilizing the local processor and not
suffering from server or network load issues. The system uses a sophisticated update process to efficiently maintain data consistency across the clients. The system presented in this thesis does not strive to just hold a local cache of a dataset, but enables knowledge based acceleration by allowing a user to open-up an algorithm and extract/insert knowledge deep into its execution, while managing the storage and transmission of the knowledge automatically.

Ghoting et al. [9] perform knowledge-conscious exploratory data mining in the domain of \( k \)-Means clustering. They first optimize \( k \)-Means clustering by using a space partitioning \( kd \)-tree to assist in eliminating unnecessary computation between iterations of the algorithm. They secondly introduce a client-side knowledge-cache; this cache holds data structure information and is incrementally updated, and the cache allows knowledge to be stored between queries for the acceleration of subsequent queries. Although similar in spirit to the Knowledge Cache proposed in this thesis, it is not a generic solution in a client/server configuration. Our Knowledge Cache permits the storage and retrieval of any data stored within a knowledge object and in a multi-client situation can share knowledge objects across them.

1.5 Organization of this Thesis

The remainder of this thesis is organized as follows. In Chapter 2 we present the Knowledge Cache, describe its architecture, implementation and runtime behavior. We then discuss the programmer’s API for use with the Knowledge Cache and knowledge accelerated data mining algorithms. Chapter 3 discusses what it means to accelerate an algorithm through the inclusion of knowledge and presents two examples examples of data mining algorithms which have been re-architected to leverage
the Knowledge Cache. In Chapter 4 we present an experimental evaluation of the algorithms presented in Chapter 3 and we make concluding remarks in Chapter 5. Finally, we include source code for the Knowledge Caching API and an example implementation of the API in Appendixes A and B respectively.
Chapter 2: KNOWLEDGE CACHING

As motivated in the introduction of this thesis we must address the need for a service to facilitate the transmission, storing, organization and retrieval of knowledge. In this chapter we will present our solution to this problem by defining the Knowledge Cache and detail its design and usage model.

2.1 The Knowledge Cache

During the knowledge discovery process a particular data mining algorithm may be executed repeatedly, with minor tweaks to the algorithm parameters to help the human hone in on the knowledge formulation they are trying to reach. From a systems perspective successive executions of a data mining algorithm in this manner can have very similar data access and computationally similar characteristics. If a programmer can identify these repeated sections they can store the computational work (the knowledge) in a Knowledge Cache, later retrieving it for re-use. If this retrieved information is relevant to the next execution of the data mining algorithm, time can be saved by finding the result in the knowledge returned by the Knowledge Cache rather than repeating computation to arrive the same place.
What knowledge can be stored is entirely dependent of the specifics of the data mining algorithm. It does require some programmer effort to expose sections of repeated computation, but in many situations considerable speedup can be realized through the incorporation of pre-computed knowledge and use of the Knowledge Cache. To the Knowledge Cache itself however there is no restriction to what knowledge can be stored; its internal representation is blind to the structure and format of the data. Information or meta-data about the knowledge is transmitted when knowledge is stored.

How is using a Knowledge Cache different from storing knowledge in a database? The Knowledge Cache is a complete solution: a service as well as a programmer’s API. The Knowledge Cache does not care about the data structure underlying the knowledge, which means any machine-representation may be used, as it’s the client who interprets the knowledge in its native form. The Knowledge Caching API provides the programmer a facility which lets the system algorithmically decide what knowledge is “best” for a instantiation of an algorithm; we call this measure the ReuseScore. How to compute the ReuseScore is entirely programmable allowing various metrics be used in making this determination, for instance, the programmer might want to favor a knowledge which closest matches the data range currently under examination. We find allowing a more natural expression of this score is more intuitive and easier to work with than having to define this score through a database’s query language. In addition to the Knowledge Caching API, the Knowledge Cache allows programmers and system designers control of how data is stored within the cluster.
The Knowledge Cache library is cross-platform and can be deployed easily on modern commodity clusters. The Knowledge Cache is highly customizable and nimble by design resulting in quick deployment with minimal systems configuration.

2.1.1 Design Goals

We now address some of the higher levels design goals which we consider components of a successful Knowledge Caching Service.

1. A Simple Abstraction

The Knowledge Cache must provide a simplistic way of conceptualizing knowledge as it is transformed from a concept to a real entity in a software system so as to impose as little intellectual burden on those responsible for re-architecting data mining algorithms to expose opportunities for knowledge re-use in conjunction with the Knowledge Cache.

2. An Intuitive Interface

To minimize integration overhead the programmers API must be intuitive and overt; the interface must exhibit program flow in a way which is natural for use. The programming sequence should be semantically similar for transmission and retrieval of knowledge, and packing of knowledge itself should be easy and flexible to suit multiple implementations of data mining algorithms.

3. Compatible

An ideal service would be compatible with modern programming languages and software libraries, whilst facilitating quick and simple deployment on modern commodity cluster infrastructures. Client to server connections should be made over standard communication libraries.
4. High Performance

The service should leverage fast modern network interconnects as well as take advantage of cluster technologies. The service should efficiently distribute workload and provide the fastest possible response time to queries and knowledge storage and removal.

5. A High Degree of Sharing

The system should support multiple concurrent connected users and facilitate sharing of knowledge across all users to maximize the likelihood of matching useful knowledge during repeated algorithm execution.

2.1.2 High Level Design Overview

The Knowledge Cache abstracts knowledge as a *knowledge object*. This object contains information about the knowledge stored in the Knowledge Cache as well as containing actual knowledge itself. The other major entity of the Knowledge Cache is the *query object*. This object is used to supply information to the knowledge cache to correctly retrieve knowledge objects. The Knowledge Cache supports two major operations which act in conjunction with the client: *Store Knowledge* and *Retrieve Knowledge*.

Knowledge objects are built by extending a template class of the API (the details of which are discussed in Section 2.2), and include information about the knowledge object (*meta-data*) along side the actual knowledge. For instance in the example of *k*-Means clustering the meta-data portion of the knowledge object may contain the dataset name, the range of the data used when clustering, the number of cluster centers used and any other fields needed to best select a knowledge object for a given
query. The knowledge portion will contain the actual data values for the centers to be transmitted to the client and used in execution. The knowledge object is also capable of reporting whether or not it can be re-used as part of a query, as well as computing a score (\textit{reuseScore}) which represents the value that object can provide if reused.

Query objects are used to submit requests for the Knowledge Cache to examine the stored knowledge objects and retrieve and transmit the one which has the highest reuse score. Query objects contain the same information as a knowledge object’s meta-data in order to perform matching.

When the Knowledge Caching service is enabled on a cluster, it is supplied with a parameter representing the number of nodes on which the service is to use as compute resources and a head node co-ordinates operations in this grouping. The Knowledge Cache receives a knowledge object for storage, a node is chosen to ‘host’ the knowledge object, and the raw knowledge portion is then split and distributed across the nodes in the group; the knowledge is stored in the main memory of each node. When a query is made to the Knowledge Cache the head node will discover the relevant knowledge objects stored across the group, rank each knowledge object by calculating its reuse score and transmit the highest scoring object back to the client. Specifics of the execution model of the Knowledge Cache is described in detail in Section 2.3. The Knowledge Cache currently employs a simple FIFO replacement policy; when space is needed (once the aggregate memory on the server nodes is consumed) the cache will evict the oldest knowledge object. Exploring other cache replacement policies is left for future work.

The Knowledge Cache has been built using freely available open source libraries and can be compiled to run on any operating system. Client/server communication
is performed over standard C-sockets and inter-node communication is performed through MPI.

![Diagram of Knowledge Cache High Level Organization]

**Figure 2.1: Knowledge Cache High Level Organization**

### 2.2 The Knowledge Caching API

The usage model is a client/server configuration with the client communicating via a point to point network connection (such as a standard C-socket) and the Knowledge Cache running on a commodity cluster accompanied by a high-speed interconnect (such as Infiniband). The organization is described in Figure 2.1.

The API models the concepts of the high level design closely (Section 2.1.2). We first describe the construction of the knowledge object, the object used to represent
knowledge in the Knowledge Cache (Figure 2.2 (left)). Following that, we detail the construction of the query object, the object responsible for encapsulating query parameters used by Knowledge Cache to select from stored knowledge objects for retrieval (Figure 2.2 (right)).

The running example in this section to illustrate details of the implementation will be Incremental Frequent Itemset Mining. Frequent itemset queries consist of a dataset name, a dataset range and a value representing the support. Queries of this sort generate knowledge in the form of Frequent Itemsets which can be used in successive queries. Incremental frequent itemset mining attempts to use existing information to mine itemsets faster. A full analysis of Knowledge Cache Accelerated Incremental Frequent Itemset Mining appears later in Section 3.3.

- **Metadata**: The metadata attribute of the knowledge object provides the identifying information about the knowledge being represented. The information inside in attribute is used in the Knowledge Cache to match a knowledge object to a query. For the case of frequent itemset mining the metadata will contain the dataset name, the dataset start and end points, and the support value that corresponds to the frequent itemsets generated during algorithm execution.

- **Bin MetadataLinearize()**: This method is used to pack the structure of the metadata into linear form (aligned sequentially in memory) used for transmission and storage by the Knowledge Cache. In our running example this would be values for the dataset name, range and support mentioned above.

- **MetadataDilinearize(Bin)**: This method takes as a parameter a pointer to linearized representation of the metadata attribute and dilinearizes it back to its
original representation. This operation is performed by the Knowledge Cache each time it needs to retrieve the metadata values. In our example this operation would restore the dataset name, range and support back into a machine usable representation.

- **Knowledge**: The knowledge attribute is an encoding of the knowledge itself. This can be any machine-representable format from a complex pointed data structure or a blob of video information. The client is the only one who sees this information in its native form and is responsible for encoding it into linear form. For frequent itemset mining the knowledge is the set of frequent itemsets generated by a query represented as a prefix tree (detailed in Section 3.3.3), their support values and a reference to the head of the tree.

- **Bin KnowledgeLinearize()**: This method is used to produce a linearized form of the knowledge mentioned in the previous bullet. The client must know how to transform the data structure into linearized form. For our example the prefix tree may be walked and copied into a sequential block of memory in visited order.

- **KnowledgeDilinearize(Bin)**: A linearized representation of the knowledge is passed as a parameter and this method dilinearizes it back to the native representation. The client must know how to decode the linearized form to restore the data structure. In the context of frequent itemset mining this would involve re-walking the prefix tree in the same order as it was linearized. Seeing as the prefix used absolute pointer addresses, the addresses must be re-offset
in observation of its new placement in memory after transmission to remain consistent.

- **canReuse**(*QueryObject Query*): This method is passed a query object as a parameter and is used by the Knowledge Cache server when making the determination as to whether the knowledge currently under examination is a match for the passed query object. It is an operation which returns a Boolean true or false with regard to the comparison. In the running example, knowledge is usable if the dataset name matches and the data ranges overlap, or if they don’t overlap, the query support is higher or equal to the knowledge support.

- **reuseScore**(*QueryObject Query*): This method is called by the Knowledge Cache server at the point where some knowledge has been determined reusable to score the reusability with respect to the query. The Knowledge Cache will return to the client the knowledge with the highest reuse score. For incremental frequent itemset mining the reuse score is calculated one of two ways: if the query dataset name and range match the knowledge object, score those with support closest to the query highest, otherwise, combine the degree to which the dataset range overlaps with the support values closest to the query into a single score. (The exact details of this operation are discussed in Section 3.3.3).

- **getIndexFields()**: This method is used for optimizing the lookup time in the Knowledge Cache server when handling storage and retrieval of knowledge objects. This method returns a list of fields and their types which may be used to index the knowledge object. In our example, knowledge objects are indexed on dataset (string) only.
The query object is implemented by extending an interface in the same fashion as the knowledge object. Again, the running example used in this section is incremented frequent itemset mining.

- **Query**: The query attribute is the encapsulation of the query parameters, the contents of which are used to match against the knowledge objects when the Knowledge Cache is selecting ones to retrieve. The query should contain similar information to the knowledge’s metadata and in the case of our example, it will store the dataset name, start and end points and the support value.

- **Bin QueryLinearize()**: This method is used to pack the structure of the query into linear form. The running example would require a packing of the dataset name, start and end points and support value.

- **QueryDilinearize(Bin)**: This method has linearized query information passed into it, and returns the dilinearized query information packed into its original data structure for use in querying knowledge objects in the Knowledge Cache by the server. Our example again calls for dataset name, range and support value.

- **getIndexFields()**: The Knowledge Cache server maintains a list of values and value types in order to facilitate quick selection of candidate knowledge objects before further pruning. The index is updated whenever a new knowledge object is added to the Knowledge Cache. Further optimization is achieved by defining a relationship vector indicating how index values of the query and knowledge object can be compared. Supported relationships are *equal to*, *less-then or equal-to* and *greater-than or equal to*. For example, the *equal* relationship says that
whatever index value the query object generates for a field may only be matched to the index value equal to those generated by the corresponding knowledge object field. This facilitates quick generation of candidate knowledge objects based on dataset name for example. The \textit{less-than or equal-to} relationship says that knowledge objects with a corresponding index value for a field less than or equal to those index values generated by the query field may be considered, and \textit{greater-than or equal to} vice versa. For our running example, the index fields are dataset name only, and the relationship is \textit{equal to} so as to retrieve knowledge objects from the same dataset as the query. We do not prune based on support or dataset range as incremental frequent itemset mining enables knowledge to be useful no matter what these values may be.

We feel that the API is intuitive and does not impose a significant intellectual or technical burden on the programmer in enabling use of the Knowledge Cache. The API could be trimmed and remain functionally equivalent with just knowledge and query object class definitions. This would require much more intervention by the programmer and does not take advantage of potential parallel server-size computation. Methods \texttt{canReuse} and \texttt{reuseScore} were developed to offload computation to the server nodes that host each knowledge object. The linearization and packing methods enable the Knowledge Cache to break apart the knowledge and store them across the main memory of the server nodes. The index fields are designed to prune unusable knowledge objects early before determining whether they are usable with respect to a query. Without these additions good scalability would not realized with the addition of extra compute nodes in the server, so the return on effort in implementing these augmentations to the API is high.
2.3 Knowledge Cache Logic Flow

As mentioned in Section 2.1.2 the Knowledge Cache supports two major operations in conjunction with the client: Store Knowledge and Retrieve Knowledge. Also, the client/server model was described. Now we present in detail how the Knowledge Cache uses the API (presented in the previous section) to support these operations. The term service and server are interchangeable. The service is started on the cluster by designating a head node and supplying a parameter which corresponds to the number of compute nodes it may utilize.

2.3.1 Store Knowledge

When the client wishes to store knowledge the following procedure is executed as follows:

1. The programmer has isolated the knowledge she wishes to store in the Knowledge Cache. MetadataLinearize() and KnowledgeLinearize() are called by the client to produce linearized versions of the metadata and knowledge attributes of the knowledge object.
2. The knowledge object is packed and a call is made to transmit the linearized metadata and knowledge to the Knowledge Cache. The head node communicates with the client via a C-socket.

3. The head node selects a node to host this knowledge object in round-robin fashion. The metadata for the knowledge object is placed on this node. The linearized block corresponding to the knowledge is then split up and distributed through the compute nodes in round robin fashion.

4. `getIndexFields()` is called by the server to retrieve the index fields, and they are stored on the node.

5. If adding this knowledge object results in the server running out of aggregate storage space, knowledge objects are deleted according to a least recently used cache replacement policy until sufficient space is available for storage of the new object.

6. The client’s is unblocked and permitted to proceed, the knowledge object is now ready for the system to query and retrieve.

### 2.3.2 Retrieve Knowledge

When the client wishes to query the Knowledge Cache and retrieve knowledge the following procedure is executed as follows:

1. A query object is instantiated by the client and its fields populated. `QueryLinearize()` is called to generate the linearized representation of the query object.

2. The query object is packed and a call is made to transmit the linearized query to the Knowledge Cache.
3. Once the head node receives this query it is immediately broadcast to all of
nodes in the service.

4. On conclusion of the broadcast, each server node calls \textit{QueryDilinearize()} to
retrieve the query object attributes for use.

5. \textit{getIndexFields()} is called by each server node to retrieve the index fields and
relationships, and each server node selects knowledge objects based on the index
to test for reusability and the reuse score.

6. For each knowledge object indexed, call \textit{MetadataDilinearize()} to retrieve the
metadata attributes for use. Then call \textit{canReuse()} to determine whether the
knowledge object is reusable, if it is call \textit{reuseScore()} to compute the score for
that node.

7. All server nodes transmit the reuse score for the knowledge objects to the head
node which decides the winning knowledge object. Ties are broken randomly.

8. The head node then gathers the chunks of the linearized knowledge from all the
cluster nodes it was distributed across and transmits it back to the client.

9. The client then restores the knowledge object by calling \textit{MetadataDilinearize()}
and \textit{KnowledgeDilinearize()} on the linearized data it just received.

The execution achieves performance in the following ways:

1. Knowledge objects are placed in round robin order across the server nodes. This
allows for fast searching of knowledge objects by first broadcasting the query to
each node, and having each node index search the knowledge objects they are
hosting and compute reusability in parallel. The number of knowledge objects stored by each server node is not controlled nor are knowledge objects moved once they are stored. There is no clear advantage of load balancing in this way as the individual parts of the knowledge object are split across the nodes regardless.

2. Knowledge is partitioned and stored in round robin fashion across the server nodes. This allows for balanced memory utilization and can better take advantage of the aggregate memory capacity (main memory and disk) of all of the server nodes. If we were to exceed the core memory size of the cluster nodes, we will enjoy the benefits of parallel disk access for maximum aggregate bandwidth.
Chapter 3: KNOWLEDGE ACCELERATED ALGORITHMS

Throughout this thesis we have prefaced our discussion with the notion that you can use knowledge to accelerate the execution of a data mining algorithm, as well as extract knowledge from the execution of an algorithm for storage and reuse in the Knowledge Cache. What does this process look like? Does it bring with it a significant investment in software re-implementation? Does it incur technical debt? In this chapter we will discuss various ways in which a data mining algorithm can be accelerated through the use of the Knowledge Cache and how to extract useful knowledge for storage and subsequent use. We will then demonstrate this process through two example data mining algorithms, $k$-Means clustering and incremental frequent itemset mining, and follow up with an analysis of their performance when accelerated by the Knowledge Cache.

3.1 Algorithm Acceleration

The central driving notion behind the Knowledge Cache is that it is faster to attain the result of a data mining algorithm’s execution through the incorporation of information extracted from previous executions of that algorithm, than it is by repeating the execution of that algorithm with no prior knowledge (from scratch).
The Knowledge Cache permits users to save a very important piece of the puzzle: state. The user may capture and store any state desired by simply augmenting it with as little identifying metadata necessary to determine whether this state can be reused in the future; the ease and flexibility at which this state can be stored is the power of the Knowledge Cache. For example, the user can store starting points for clustering algorithms centers, node labels for graph data structures all the way to complex data structures like hyper-graphs, tensors, memoization tables etc. The Knowledge Caching API allows this knowledge to be directly “plugged into” program logic and used immediately without any required preprocessing beyond decoding or delinearizing the data structure. This knowledge can then be used in various ways to short-cut the execution of an algorithm. It can provide the results of calculations, or it can be the result of traversals of data structures, or it can be parameter initializers to reduce the overall number of algorithm iterations. In each case the user reuses knowledge to speed up execution by avoiding repeating the utilization of compute, storage and data transfer resources.

Are all algorithms available for acceleration through use of the Knowledge Cache? That is not a straightforward question to answer, but there are some guiding principles that we feel could assist programmers making this determination.

- Output: If a data mining algorithm is executed with a particular set of parameters and the output is similar to that of another execution of the same algorithm with slightly altered parameters then this property can be exploited. The knowledge in this instance can be the result of the query and the programmer could develop a post processing step which can derive the result from the knowledge directly without having to repeat execution of the algorithm [23].
• Transitions: If there is an identifiable pattern of how state changes in the execution of an algorithm then this property could be exploited. For instance, if in the course of the execution some large matrices in a known state can have some complex operation performed on them, or some grid of coefficients are generated based on other known states, and this can be concisely described through metadata, then this can be stored in the Knowledge Cache. Each time the algorithm needs to transition between states the Knowledge Cache could be consulted and the knowledge retrieved to accelerate the completion of the algorithm [18].

• Iteration: If a data mining algorithm iterates and converges as part of its execution then this property can be exploited [9]. The knowledge in this case would be the convergence values reached by a previous execution of the algorithm with similar parameters. If the start points are initialized from knowledge retrieved from the Knowledge Cache then total iterations may be reduced.

• Sampling: If a data mining algorithm can operate on a sampling principle then the knowledge object can act as a sample. Often in algorithms that use sampling, obtaining a good sample is the hardest part in achieving good performance [28]. If a sample is stored in the Knowledge Cache and the metadata can describe what that sample represents, it could very well be the key to realizing good accuracy and performance.

The above principles can assist users identifying opportunities for algorithm acceleration through knowledge reuse, but are not intended to restrict usage models. The Knowledge Caching API is flexible enough to extend many novel usage models. If
the algorithm under examination possesses some of the characteristics outlined above then it is very likely that there will be gains through the use of the Knowledge Cache.

We now present two full example implementations of knowledge accelerated data mining algorithms which use the Knowledge Cache: $k$-Means clustering and incremental frequent itemset mining. For each, we provide algorithmic background, insight into how we enabled the algorithm for acceleration through the use of the Knowledge Cache and provide specific details of how we used the Knowledge Caching API in the implementation.

3.2 $k$-Means Clustering

3.2.1 Preliminaries and Background

The $k$-Means clustering algorithm [16] is a classical data mining algorithm used in cluster analysis. The aim of the algorithm can be efficiently described as dividing $n$ data points in $d$ dimensions into $k$ clusters such that the within-cluster sum of squares is minimized. The algorithm does not find an optimal solution, but instead a local minima. A naïve implementation can be defined as follows. First, initialize $k$ random centers, $C^0 = \{C_1^0, \ldots, C_k^0\}$. Next, for each of the $n$ data points, finds its closest center in $C^0$. Partition the data points into $k$ subsets based on their closest centers. Compute the center of mass for each of these $k$ subsets, these become the new set of $k$ centers $C^1 = \{C_1^1, \ldots, C_k^1\}$. Repeat this process iteratively until an iteration $i$ is encountered such that the centers $C^i$ and $C^{(i+1)}$ are identical.

We can see that for each iteration of the algorithm, there must be $n \times d \times k$ ownership checks followed by the recalculation of the cluster centers’ positions. The algorithm will iterate until the centers rest at their local minima (dependent on their initialized
values), with the final output being $k$ centers $C_i$. This iterative characteristic may be exploited and the algorithm may be accelerated through the inclusion of knowledge. If we can reduce the number of iterations by better initializing the values of $C^0$, the savings in computation time can be considerable. It has been demonstrated in [4, 12] that if we possess the results for a previous query that processed same same dataset on an overlapping data range as a new query, the centers may be initialized to those values allowing the execution of the new query to converge faster. Intuitively, if two queries have data ranges which overlap then the overlapping region has influence on the centers, pulling them closer towards each other. As such, the algorithm is accelerated through the inclusion of knowledge from previous queries to reduce the total number of iterations by initializing centers of similar queries closer to each other. We now detail how to build a complete knowledge accelerated solution which delivers this information.

### 3.2.2 Application of the Knowledge Cache API

1. The Knowledge Object:

   First the user must determine specifically what it is that will represent knowledge in the case of $k$-Means clustering, in this case it is an array of $k$ floating point values corresponding to $k$ cluster centers. Knowledge represented in this form will be easy to construct and pack into the knowledge object, as well as extract and plug back into the algorithm. Referring back to Figure 2.2 (left) the user must create a knowledge object, the API provides a template class for the user to extend and implement.
The purpose of the knowledge object is to programmatically encapsulate the knowledge and store information about how to automatically select and rank this object at query time. We will populate these fields with customized information specific to this implementation of knowledge accelerated k-Means clustering. The metadata member of the knowledge object will be composed of an encoding of the dataset name, the number of centers the start point and the end point. The metadata is then linearized by placing each of these values in a contiguous block of memory and augmenting the structure with some size information used in delinearization. The knowledge member is an encoding of the $k$ center values, linearized by building an array. We must complete the knowledge object by identifying the index fields which the Knowledge Cache will use to quickly example candidate objects. In this case the index fields are the data set name and the number of centers. We do not select the data range as index fields because query and knowledge object range values need not be exact; their similarity will be used when generating the reuseScore metric.

The knowledge object also allows the user to instruct the Knowledge Cache how to determine whether a candidate knowledge object can be reused as well as how to rank the knowledge objects with their usefulness with respect to the query object. This is achieved by implementing virtual canReuse() and reuseScore() methods. canReuse() is used as a simple Boolean check to determine if a candidate knowledge object is reusable. In our example, the dataset name must match and the number of centers much be greater than or equal to the query. In addition, the data range start and end points must overlap. In implementing the reuseScore() method, the user may programmatically rank
reusable knowledge objects against a query and quantify how much ‘usefulness’ the knowledge contained within it can apply to satisfying a query. In this example we chose a two-part score with each component contributing up to 100 points for a reuseScore in the range of 0-200. The object was to rank knowledge objects with the closest signature to that of the query; score highly if the number of centers match closely, and score highly when the data range overlays most closely. The scoring is as follows ($\frac{Knowledge\_Range\_Overlap}{KO\_End\_Range - KO\_Start\_Range}$) for the data range component + $\frac{NumQueryCenters}{NumKnowledgeObjectCenters}$ for the number of centers component all, multiplied by 100. This score is not necessarily optimal, ranking results an exhaustive topic. This score allows for knowledge objects with vastly different centers and range overlap signatures to both rank highly. The main point here is that the user can have control over how the knowledge objects are ranked and may tune the ranking to better suit their needs. In our experimentation we found that broad, general reuseScore metrics provided sufficient quality knowledge objects for knowledge accelerated algorithms.

2. The Query Object: The query object is used to provide to the Knowledge Cache a query with which to select candidate knowledge objects for ranking, selection, then transmission back to the client. The query object is a straightforward part of the API to implement, shown in Figure 2.2 (right). The user must populate the query member of the query object in the same fashion as the metadata in the knowledge object. In our example this will again be an encoding of the dataset name, the number of centers the start point and the end point. We must also set up the fields pertaining to lookup optimizations; the index fields will again be the data set name and the number of centers. An additional
optimization in of the query object are the index relationship fields (the fields which enable early elimination of candidate objects on the Knowledge Cache): 

/EQUAL_TO/ for the data set field as we can only use knowledge objects from the same dataset, and /GREATER_THAN_OR_EQUAL_TO/ for the number of centers field as we are interested in receiving back knowledge objects with greater or the same number of centers as what was requested in the query.

The approach is consolidated in Algorithm 1.

**Algorithm 1** Knowledge Accelerated /k/-Means clustering

\[
C^0 = \text{numCenters} \text{ random centers} \\
q = \text{Query(dataset, numCenters, startRange, endRange)} \\
q = q.\text{QueryLinearize}() \\
\textbf{if} \ ko = \text{RetrieveKnowledge}(q) \ \textbf{then} \\
\quad ko = ko.\text{KnowledgeDilinearize}() \\
\quad C = k\text{Means}(ko.\text{Knowledge}, \text{numCenters}, \text{startRange}, \text{endRange}) \\
\textbf{else} \\
\quad C = k\text{Means}(C^0, \text{numCenters}, \text{startRange}, \text{endRange}) \\
\textbf{end if} \\
k_{\text{new}} = \text{KnowledgeObject()}
\]

\[
k_{\text{new}}.\text{Knowledge} = C \\
m = \text{Metadata(dataset, numCenters, startRange, endRange)} \\
m = m.\text{MetadataLinearize}() \\
k_{\text{new}}.\text{Metadata} = m \\
k_{\text{new}}.\text{KnowledgeLinearize}() \\
\text{StoreKnowledge}(k_{\text{new}})
\]

The knowledge accelerated /k/-Means clustering algorithm will query the Knowledge Cache upon execution. If a usable knowledge object is found the knowledge is extracted and used to initialize the /k/- centers; if no usable knowledge object is found the /k/- centers remain random. Finally a new knowledge object is built and linearized and stored in the Knowledge Cache through a /StoreKnowledge/() call. In
our running example, a usable knowledge object can also be returned to the user with \textit{numCenters} greater than \( k \). If this is the case, we hierarchically cluster [14] the centers by merging nearest centers (on the Euclidean measure) until we are left with \( k \).

Association rule mining follows as a more involved application of algorithm acceleration through the use of the Knowledge Cache. The algorithm required rework in order to incorporate knowledge and the knowledge itself took on a more complex form by way of being an entire data structure rather than an array of initializations as previously described.

3.3 Incremental Frequent Itemset Mining for the Generation of Association Rules

3.3.1 Preliminaries and Background

Association rule mining is a method for discovering relations between variables in large databases. It is another instance of a well researched and classic data mining algorithm originally developed for determining relationships between items in a large scale transactional database [1]. The idea is that customers, say, at a grocery store purchase certain items in sets for example, burger buns, onions, and ground beef. These items can imply rules. For example, if a customer buys burger buns and onions, they are likely to also buy ground beef. This information can be used as the basis for marketing promotions and placement of products in the store. If many customers who buy burger buns, onions and ground beef also buy mustard, then a mustard coupon may print the next time the customer makes their purchase. The first step in this process however is generating these sets of commonly co-occurring items in transactional data and this process is termed frequent itemset mining, and
is the algorithm at the focus of this example. First we offer brief preliminaries to describe the formation of the problem, then describe how we used the Knowledge Cache to accelerate a frequent itemset mining algorithm.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of $n$ items and let $X$ be a non-empty ordered set of items with $X \subseteq I$. $X$ is called an itemset, and for an itemset of length $k$, we call this a $k$-itemset. A transaction $T$ is a set of items $T \subseteq I$ uniquely identified by their transaction ID, or $tid$. The set of transaction IDs for which contain a given itemset, for instance $X$, is called the tidlist and is denoted as $\tau_D(X)$. The percentage of transactions in a dataset $D$ which contain $X$ as a subset is called the support and is denoted $\sigma_D(X)$ and is computed as $\sigma_D(X) = |\tau_D(X)| / |D|$. An itemset is frequent if its support is no less than a user specified support threshold $\text{minSup}$, i.e. $\sigma_D(X) \geq \text{minSup}$. An associate rule is an implication of the form $X \Rightarrow Y$, where $X,Y \subseteq I$ and $X \cap Y = \emptyset$. Association rules also carry a support and a confidence value. The support is the joint probability of $X$ and $Y$ i.e., $\sigma_D(X \cup Y)$ whereas the confidence is the conditional probability of $Y$ given $X$ i.e., $\sigma_D(X \cup Y) / \sigma_D(X)$. An association rule is confident if its confidence value is no less than a user specified $\text{minConf}$ value. Once the frequent itemsets are generated from a dataset, the association rules are enumerated for all values and their confidence is calculated [29].

In this example we are not interested in generating association rules and their confidences as it is a straight-forward exercise, the challenging portion is discovering the frequent itemsets in transactional data and it is this aspect we intend to accelerate through the use of the Knowledge Cache. Extensive research on this problem has produced a variety of frequent itemset mining algorithms which operate on a several core methodologies [26]. Algorithms such as Apriori [2] operate by generating candidate
itemsets of increasing length, testing their support and pruning infrequent itemsets at each step. Algorithms such as FP-Growth and Eclat [15, 31] do not generate candidate items but instead only grow frequent itemsets; FP-Growth uses a novel data structure to do this while Eclat exploits properties of a vertical database layout (tidlists). There are other methodologies too, such as constraint based mining, approximate pattern mining, sampling based mining and methods which reduce the total number of patterns generated whilst still finding all patterns (GenMax) [21, 22, 28, 13].

3.3.2 Designing for the Knowledge Cache

There are properties of frequent itemset mining which can work to our advantage. For instance the frequent itemsets generated from a dataset with support $minSup_1$ will also be present in those generated from the same dataset with a lower $minSup_2$ value. This is intuitive, if an itemset $X$ has support $\sigma_D(X) \geq minSup_1$, and $minSup_1 > minSup_2$ then $\sigma_D(X) > minSup_2$. What this means is that if the frequent itemsets are stored in the Knowledge Cache for a certain dataset and support threshold, a subsequent execution of a frequent itemset mining algorithm can use this information and generate some frequent itemsets for ‘free’. We would however like the implementation to be more flexible than just being able to make use of knowledge objects with the same dataset and lower support values as this will only make a limited number of knowledge objects available to use for acceleration. Ideally we’d like any knowledge object to be useful to a query provided that the frequent itemsets contained within were generated from the same dataset. This means that our frequent itemset mining algorithm must be able to tolerate knowledge objects containing itemsets with higher support than the query that is being processed. Another degree
of flexibility necessary to enable use of a greater number of knowledge objects would be the ability to perform frequent itemset mining on dataset ranges or subsets, in a similar fashion to that of \( k \)-Means clustering. This carries additional implications for our frequent itemset mining algorithm as it now must be able to process knowledge objects generated over a different data range than that of the query, introducing false positive frequent itemsets as well as the absence of itemsets which may be frequent in the query’s range. We will have to develop canReuse and reuseScore metrics to only select the best knowledge objects for this application, but by allowing as many knowledge objects to be considered by the Knowledge Cache as possible we stand to achieve the highest amount of benefit. In order to make use of the inexact knowledge objects the frequent itemset mining problem must be re-thought to develop a general solution, when given a knowledge object containing itemsets generated from a data range and or support different from the current query it must be able to discover new frequent itemsets, reject itemsets which are no longer frequent and recalculate the support counts to reflect the actual result.

The approach taken in this example is to re-model it as an incremental frequent itemset mining problem. Incremental mining was first introduced in [6] and its central idea is to re-use previously mined itemsets to avoid re-computation and expensive database scans. This is accomplished by tracking which data was added and removed between executions of the algorithm and using the information contained within the differences to make decisions when mining itemsets; these deltas are typically smaller than the dataset allowed for reduced total computation.

For this discussion we adopt the same notation as [29]. Let \( minSup_D \) be the minimum support threshold used when mining \( D \), and \( L_D \) be the set of frequent
itemsets generated. Let $P$ be the information retained from the current mining operation which will be kept and applied to the next mining operation, $P$ in our case will be a set of itemsets as well as the 1-item support counts. There has been an incremental change to $D$ when a new set of transactions $d^+$ have been added and a set of transactions $d^-$ have been removed. This forms a new set of transactions and a new dataset $\Delta$, with $\Delta = (D \cup d^+) - d^-$. Modifications can be considered a special instance of a transaction deletion followed by an addition so these need not be considered here. The problem now of incremental frequent itemset mining is to find the set $L_\Delta$ which are the frequent itemsets in $\Delta$ with $minSup_\Delta$, while incorporating $P$ to achieve higher efficiency. As transactions are added ($d^+$) and removed ($d^-$) from $D$ to form $\Delta$, certain frequent itemsets may no longer remain frequent and itemsets which weren’t frequent may become frequent. We call these declined and emerged itemsets respectively. If a frequent itemset remains frequent in both $D$ and $\Delta$ we call this a retained itemset.

This model works well with the knowledge cache. Each execution of a new frequent itemset mining query can now be thought of as an instance of incremental frequent itemset mining, with $P_D$ being the set of frequent itemsets obtained from a previous execution of the algorithm on a dataset $D$ with a known $minSup$ stored in a knowledge object and provided by the Knowledge Cache. $d^-$ and $d^+$ now become the difference in the data ranges between the query and those in the knowledge object; determining these sets will be a pre-processing step. We will use tidlists to represent the dataset in our example as this affords quick and simple support counting. According to [31], any itemset can be obtained by joining its 1-items, and the support can be obtained by intersecting the tidlist of each 1-item. By constructing the tidlists
ourselves we have full access to the tidlists of $D, d^-$ and $d^+$ enabling us to compute the support of any itemset in $\Delta$.

Finally, we must select an algorithm and determine how specifically we are going to incorporate knowledge for algorithm acceleration. What the Knowledge Cache is providing for us in this example is $P_D$ which are the frequent itemsets and support counts on $D$ above a certain $minSup_D$. This means we do not have to re-discover these itemsets as long as they are still frequent in $\Delta$, and computing the value of $\sigma_\Delta(X)$ can be done using only the knowledge object and $d^-$ and $d^+$. The way we solved this problem was to use the knowledge object as an initialization of the result of $L_\Delta$, then we would ‘grow’ this initialization by discovering new frequent extensions of the existing itemsets until the complete result was found. The algorithm operates in similar spirit to ZIGZAG [29], which uses a set of frequent extensions (called the combine set) to grow and test existing itemsets and retain grow and test emerged itemsets by traversing the search space of new itemsets. It performs a backtrack-style traversal of each newly discovered itemset, continuing the search on every item which remains frequent. Our algorithm will instead walk the tree contained with the knowledge object discovering new emerged itemsets at each level, adding them to the tree for growth later on in the traversal. It has the advantage that discovery of new itemsets can end sooner when all possible extensions ‘run out’ for a particular itemset prefix and it will never perform a support calculation on a candidate emerged itemset more than once. The algorithm is explained in more detail below.

The algorithm makes use of the trie data structure, which is the encoding used to represent the set of itemsets in a knowledge object. The rationale and design choices supporting this data structure are discussed in more detail in section 3.3.3, but the
important property to bear in mind for now is that each path along the trie stores a
sequence with the same prefix along a path up to a node, branching into a new node
when the sequence differs. The support is stored at the last node in the path which
makes up that itemset, and there are no duplicate transactions stored in the tree.
The algorithm works in two phases: phase one post-processes the knowledge object
and builds a new trie which contains only the itemsets which are still frequent for
the query being processed and adjusts their support counts, and phase two walks the
new trie and grows each retained itemset to discover emerged itemsets.

Phase one is simply a depth-first traversal of the knowledge object. For each
itemset \( I \in L_D \), compute \( \sigma_\Delta(I) = \sigma_D(I) + \sigma_{d^+}(I) - \sigma_{d^-}(I) \). This step uses \( \tau_{d^+}(I) \) and
\( \tau_{d^-}(I) \) to compute support much more quickly than accessing \( \tau_\Delta(I) \) as the incremental
change to the dataset is typically small (this is guaranteed by choosing only knowledge
objects which have a high degree of overlap with the range of the query). Phase two
first generates a set of frequent extensions called the \textit{combine set}, a term borrowed
from [13]. The combine \( C_l \) set is a subset of \( I \) which contains the frequent extensions
of itemset of length \( l \). \( C_0 \) initially contains the frequent 1-items and their support
counts in \( \Delta \). Each one of these 1-items is added to the new trie as new 1-itemsets
(if they are not already contained within it as such), which form new path prefixes
to be grown while the tree is being traversed. As the traversal proceeds, an itemset
prefix is formed from the path of the root of the tree up to the current node. Test the
itemset prefix against each item in \( C_l \), if an emerged itemset is found add it to the
trie, while storing the item from \( C_l \) which generated the emerged itemset in a new
set \( C_{l+1} \). Seeing as no frequent itemset can have an infrequent subset, it is valid to
only test items in the subtree of the current node with the frequent extensions \( C_{l+1} \).
at this point. Recurse on the children and siblings of the current node, sending $C_{l+1}$ as the new combine set $C_l$, and stop recursing when the $|C_l| = 0$.

It is worth reminding the reader at this point that we will not incrementally mine the datasets if the query dataset and data range are the same as the knowledge object but the query support is higher than the support of the knowledge object returned by the Knowledge Cache. As mentioned at the beginning of this section, the result of the query in this case would be a subset of the knowledge returned by the Knowledge Cache thus obtaining the results for the query can be a separate post-processing step akin to phase 1 described above. i.e. for each itemset $I$ in the knowledge object we simply check whether or not $\sigma_D(I) \geq minSup$. The pseudocode for the knowledge accelerated incremental frequent itemset mining algorithm is detailed in Algorithms 2 and 3.

**Algorithm 2** Phase1($root, I_l, newTrie, l$)

$I_{l+1} = I_l \cup root.itemName$

$\sigma_\Delta(I_{l+1}) = root.support + \sigma_{d+}(I_{l+1}) - \sigma_{d-}(I_{l+1})$

if $\sigma_\Delta(I_{l+1}) \geq minSup$ then

newTrie.addTransaction($I_{l+1}$)

end if

Phase1($root.child, I_{l+1}, this, l + 1$)

Phase1($root.sib, I_{l+1}, this, l + 1$)

3.3.3 Application of the Knowledge Cache API

1. The Knowledge Object:

To facilitate building the knowledge object for the context of this example we must first describe how exactly we are going to represent the knowledge to make
Algorithm 3 Phase2(root, I_l, newTrie, C_l, l)

if |C_l| = 0 then
    return;
end if
C_{l+1} = ∅
I_{l+1} = I_l ∪ root.itemName
P_{l+1} = \{ y | y ∈ C_l ∧ y > root.itemName \}
for x ∈ P_l do
    I_{l+1} = I_l ∪ \{ x \}
    if σ_Δ(I_{l+1}) > minSup then
        C_{l+1} = C_{l+1} ∪ \{ x \}
        newTrie.addTransaction(I_{l+1})
    end if
end for
Phase2(root.child, I_{l+1}, this, C_{l+1}, l + 1)
Phase2(root.sib, I_{l+1}, this, C_{l+1}, l + 1)

it both efficient to encode (linearize), decode (dilinearize), and access during execution. We are going to need to store a list of frequent itemsets along with their support values as well as information about the dataset, data range and support threshold which corresponds to this result. This metadata information will be stored in the metadata portion of the knowledge object, we will encode the dataset name as a string, while using integers for the data start range, end range and support threshold. The metadata is linearized by placing each item in contiguous memory and appending it with some size information.

The data structure we will use to store and access the list of itemsets is a modified prefix tree or trie. A trie has the property that each path along the tree stores a sequence with the same prefix up to a node, branching into a new node when the sequence differs. This allows compression of the itemsets by re-using paths for itemsets with the same prefix. We append each node with
the support value for the itemset which makes the path up to that node. Seeing as there will be no duplicate entries in the list of frequent itemsets, we can say that if there is a non-zero support value at that node an actual itemset was found. A value of zero means that itemset is not an actual frequent itemset in the knowledge object. The tree is built by following the prefix path for a new itemset (if it exists) and appending the tree with the remainder of the itemset when the common prefix ends. The Knowledge Cache API allows for data structures to be encoded in their runtime state meaning that once the structure is dilinearized from the knowledge object it should be usable without further translation, provided only that the structure be representable in a linearized contiguous block of memory. Although the trie is a dynamic data structure it may be dilinearized by performing a depth-first traversal and copying nodes into an array of trie nodes and adjusting their pointers to be relative to the base address of the array. A further advantage of this is that once the knowledge is returned by the Knowledge Cache the trie can be reused immediately after the points have been re-adjusted for their new place in memory. Figure 3.1 is a visual representation of a trie which has encoded a set of itemsets, along with its linearized form. The index field for this example is only the dataset name, we do not wish to index the data range or support value as we wish to consider all knowledge objects and can choose the ones with the best chance for introducing useful knowledge through the `reuseScore()` method.

The `canReuse()` and `reuseScore()` methods are the final portions of the knowledge object to implement. Again, these are used by the Knowledge Cache when determining which knowledge object to send back in response to a query. We
want to make as many knowledge objects as possible available for consideration, but we must also rank each knowledge object by the amount of valuable content it can bring to the frequent itemset mining algorithm. Remember, the knowledge accelerated algorithm is going to use knowledge to avoid recalculating an itemset’s support value from scratch (over \( \Delta \)). \textit{canReuse()} will take the form of two simple criteria: that the datasets match and there is overlap. This means we desire to provide a knowledge object with the closest matching result to that of the query. There are many factors at play when attempting to make this determination: the distribution of itemsets throughout the dataset, the support value used and the degree of overlap between the knowledge object and query range, a slight modification to one of these parameters can introduce significant differences. We want to strike a balance between the degree of overlap, and the similarity of the support values, favoring those which can provide the most useful information. The \textit{reuseScore()} has a value between 0-101. 50 points comes from a \textit{range score} and 50 points comes from a \textit{support score}, with a special case providing all 101 points. If the query and knowledge object range are the same, and the query support is higher or equal to the knowledge object support, then the answer to the query is contained totally within the knowledge object and is weighted with the highest possible score of 101. The \textit{range score} is 
\[
\min((\frac{\text{Knowledge\_Range\_Overlap}}{\text{KOEndRange}-\text{KOStartRange}}) \ast 50, 50),
\]
the percentage of overlap, capped at 50 if the range of the knowledge object is ‘wider’ than the query. The \textit{support score} is 
\[
\left|\frac{\max(\text{querySupport},\text{koSupport})-\text{querySupport}}{\max(\text{querySupport},\text{koSupport})}\right| \ast 50.
\]
This provides equal opportunity for the data overlap and support values to contribute to the reuse score.
2. The Query Object:

The query object is used to provide to the Knowledge Cache a query with which to select candidate knowledge objects for ranking, selection, then transmission back to the client. The user must populate the query member of the query object in the same fashion as the metadata in the knowledge object. We encode the dataset name, start range, end range and support value in a contiguous block of memory. We must also set up the fields pertaining to lookup optimizations; the index field will again be the dataset name with the relationship $EQUAL\_TO$. 

Figure 3.1: A Trie Encoding of Frequent Itemsets and the Linearized Form
The approach is consolidated in Algorithm 4. The Knowledge Accelerated Frequent Itemset Mining will query the Knowledge Cache upon execution. If a usable knowledge object is found the data range and support is checked. If the query and knowledge object data ranges are the same then the support is checked, and if the query support is higher or equal to the knowledge object support, then the query results is completely contained within the knowledge object and extracting these is a simple traversal of the trie. If this is not the case, the tidlists for $d^+$ and $d^-$ are generated and a calls to Phase1 and Phase2 are made. If no valid knowledge object is found in the cache, a backtrack-search frequent itemset mining algorithm is executed which operates identically to that of [13] getting the support counts from $\Delta$ directly. The tidlist for $\Delta$ is built with one necessary scan of the dataset.
Algorithm 4 Knowledge Accelerated Frequent Itemset Mining

\[ newTrie = new\ trie() \]

\[ L_\Delta = \emptyset \]

\[ C_1 = \text{read frequent 1-items} \]

\[ q = \text{Query}(\text{dataset, minSup, startRange, endRange}) \]

\[ q = q.\text{QueryLinearize}() \]

\[ \text{if } ko = RetrieveKnowledge(q) \text{ then} \]

\[ ko = ko.\text{KnowledgeDilinearize}() \]

\[ \text{if } (q.\text{Range} = ko.\text{Range}) \land (q.\text{minSup} \geq ko.\text{minSup}) \text{ then} \]

\[ L_\Delta = \text{itemsets from } ko \text{ with support } \geq q.\text{minSup} \]

\[ \text{else} \]

Build tidlists for \( d^+, d^-, \Delta \)

\[ \text{Phase1}(ko.\text{root}, \emptyset, newTrie, 0) \]

Add items from \( C_1 \) to \( newTrie \) to form new 1-itemsets

\[ \text{Phase2}(newTrie.\text{root}, \emptyset, newTrie, C_1, 0) \]

\[ L_\Delta = newTrie \]

\[ \text{end if} \]

\[ \text{else} \]

\[ L_\Delta = \text{Backtrack\_Search}(\emptyset, C_1, 0) \]

\[ \text{end if} \]

\[ ko_{new} = \text{KnowledgeObject()} \]

\[ ko_{new}.\text{FreqItemsets} = L_\Delta \]

\[ m = \text{Metadata}(\text{dataset, minSup, startRange, endRange}) \]

\[ m = m.\text{MetadataLinearize}() \]

\[ ko_{new}.\text{Metadata} = m \]

\[ ko_{new}.\text{KnowledgeLinearize()} \]

\[ \text{StoreKnowledge}(ko_{new}) \]

---

44
Chapter 4: EXPERIMENTAL EVALUATION

We now evaluate the performance and runtime characteristics of the Knowledge Caching Service as a whole through the experimental evaluation of the two data mining algorithms we’ve used as running examples throughout this thesis: \( k \)-Means clustering and frequent itemset mining. The algorithms were implemented as described in the previous chapters with the use of the Knowledge Caching API. The test-bed for this evaluation is a commodity cluster of 64 compute and 5 storage nodes with each node containing dual socket AMD Opteron-250 2.4Ghz 64-bit single core CPUs with 1MB L2 cache, and 48-bits of addressable virtual memory with 8GB of physical memory. Unless otherwise stated, a “node” consists of single a machine in the cluster executing the test code in a single thread on a single processor (the machine can utilize the other processor for kernel tasks). The cluster runs the 2.6.18-92.el5 Linux Kernel. Infiniband is used as the high-speed interconnect and uses MPI for message passing while including 1Gbps Ethernet for TCP/IP; currently the client connects to the server via a POSIX TCP/IP socket.

For \( k \)-Means clustering we generated synthetic datasets for experimentation. The datasets are built by generating 100 centers with 1000 points for each center based on a Gaussian distribution with mean equal to the cluster center with standard deviation 1; we generated data in 2, 4 and 16 dimensions. For frequent itemset mining we used
a suite of datasets from the FIMI repository [11], which include: accidents, chess, connect, kosarak, mushroom, pumsb, pumsb*, retail, T10I4D100K, T40I10D100K and webdocs.

We defined a workload as a list of \( n \) queries to be processed in order, and we developed an automatic query generation engine to produce query files which would be read and processed by the client. Query files are of the format: \([\text{<dataset>} \ \text{<data range start>} \ \text{<data range end>} \ \text{<num centers>]}\) (for \( k \)-Means clustering), and \([\text{<dataset>} \ \text{<support>} \ \text{<data range start>} \ \text{<data range end>]}\) (for frequent itemset mining). In the absence of a standard query workload for a data mining system of this sort we developed four workload types which aim to mimic realistic query patterns of users during knowledge discovery. The four workload types are denoted as follows: Random, Structured30, Structured60 and Structured90. In Random, each query is not dependent on one another and has equal chance of being different than the last, i.e. any of the attributes can change by any amount between successive queries in the query file. Structured\( X \) aims to mimic more real-world interactive data mining behavior, in which a user may be submitting multiple similar queries to the system but only changing support and data range values while keeping the dataset the same. This would be the case if the user was interactively data mining a data set, digging down to the information they seek by repeatedly submitting more refined queries at each iteration. Queries then become a ‘minor variation’ of the last i.e. the dataset will remain the same, but the data range and support value will be changed by a small amount. In a Structured\( X \) workload, there is an \( X\% \) chance that the next query will be one of these minor variations, and a \( (1 - X)\% \) chance that the query will be independent with new values for each parameter. Query files are
generated automatically based on the above specifications with 100 queries generated for the $k$-Means clustering query files and 40 queries being generated for the frequent itemset mining query files.

The following experimental evaluation intends to examine the actual performance increase gained by the use of knowledge accelerated algorithms and the use of the Knowledge Cache verses un-accelerated algorithms that don’t. First, we examine the performance characteristics of a single client across each of the workload types. We compare the performance without the Knowledge Cache to that of a configuration which uses a Knowledge Cache running on one, two and four processor nodes. Secondly, we examine the effect of permitting 4 clients ‘share’ a single Knowledge Cache in various combinations.

We must stress that we did not set out to outperform the state of the art implementations of these algorithms, as this is largely orthogonal to the contributions of this thesis, instead we wanted to show that gains from the Knowledge Cache are algorithm independent, we wanted to treat the algorithms like a black-box and test the result of accelerating them through the inclusion of knowledge and the use of the Knowledge Cache.

4.1 Results

For $k$-Means clustering and frequent itemset mining we present results for our experimental evaluation. For each algorithm we experimented using workloads as described above on various client/server configurations. We first tested a single client using no Knowledge Cache, then using a Knowledge Cache comprised of one, two and four processor nodes in order to examine the effect of making more processing
and aggregate memory resources available. We then tested a four client configuration using no Knowledge Cache, then using single Knowledge Cache processor node per client, a single Knowledge Cache processor node shared between the clients, then 4 Knowledge Cache processor nodes shared across the four clients. The goal of this experiment is to examine the Knowledge Cache performance under a multi-client environment and seeing as each of the 4 clients can access any knowledge object stored on the Knowledge Cache, we can test whether sharing nodes across the clients can yield a further benefit.

4.1.1 \textit{k}-Means Clustering

\textbf{Single Client, Multiple Servers}

Figure 4.1 shows the execution times for each of the four workload types on each client/server configuration. Each cluster of bars represents a workload type, and each bar from left to right represents an increase in Knowledge Cache processor nodes from 0 in the no Knowledge Cache case, through one, two and four Knowledge Cache processor nodes. Examining the data we can see that there is a significant speedup in the range of 1.82x (for Structured60) - 3.18x (for Structured90) when we introduce the Knowledge Cache. This is due to the enhanced initialization of the cluster centers based on knowledge retrieved from the Knowledge Cache. In this case the centers were moved closer to the true centers for this query and the algorithm converged faster. We also see small gains when the number of server nodes is increased from one to two. Doing this adds additional aggregate main memory across the Knowledge Cache coupled with parallelization of retrieving the knowledge object’s data for transmission to the client. There is no appreciable gain, and in the case of Structured60, a slight decrease in performance when increasing the server nodes from two to four; this is
most likely due to the fact that the knowledge objects generated in \( k \)-Means clustering are small and do not fill the memory of this server configuration. We can also note at this point that a StructuredX workload can have an impact on performance. When there is a high probability that a query will be similar to the previous one, there is a high chance that the knowledge retrieved previously can be applicable to that query. This is demonstrated in when comparing the Structured30 and Structured 90, as well as Structured60 and Structured90 workloads; between these you can observe a definite reduction in processing time brought on by the higher knowledge object reuse.

![Bar chart showing processing time (in seconds) for \( k \)-Means clustering workloads, single client, varying server nodes.](image)

Figure 4.1: Processing time (in seconds) for \( k \)-Means clustering workloads, single client, varying server nodes.
Multiple Client, Multiple Servers

Figures 4.2, 4.3, 4.4, 4.5 show the execution time of Random, Structured30, Structured60 and Structured90 workloads respectively when executed on a four client configuration. Unique query files were generated for these experiments. Within each bar cluster from left to right, we show the execution times under four server configurations: no Knowledge Cache, one server node per client with no sharing, one server node between all four clients shared, and four server nodes between all four clients shared. Again we see immediately see the benefit from developing the Knowledge Cache for the same reasons discussed in the previous section. What is of further interest to us is observing that when we switch from a case where the Knowledge Cache is not shared between the clients to one where it is, we see a further reduction in processing time. This indicates to us that knowledge obtained from an algorithm execution on a certain node can be relevant and useful to another node. Four clients contributing knowledge objects to a communal pool allows for a higher chance of a query matching a knowledge object with a higher reuseScore, as was one of the objectives of introducing the knowledge cache. There are instances in which the transition from two to four shared server nodes resulted in a degradation in performance. This is again a result of the knowledge objects that are generated in frequent itemset mining being small and not consuming a large quantity of the aggregate memory across the server nodes and the suffers from additional overhead of the coordination between four nodes.
Figure 4.2: Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Random workload.

4.1.2 Incremental Frequent Itemset Mining

Single Client, Multiple Servers

Figure 4.6 shows the execution times for each of the four workload types on each client/server configuration for the frequent itemset mining case in the exact fashion as section 4.1.1. Again we see speedup for all workloads with significant speedup in the range of 2.57x (for Structured60) - 2.98x (for Structured90) when we transition from no-Knowledge Cache to a single Knowledge Cache node. When the Knowledge Cache is used during workload processing, knowledge objects start to occupy the cache providing possible matches to queries submitted later in the workload. The knowledge accelerated frequent itemset mining algorithm receives a partial solution and is able
to extract the retained itemsets and grow them to find emerged itemsets faster than mining from scratch. The results do not indicate a benefit from increasing the number of server nodes from one to two or four in this experiment. This is due to the workloads not filling the aggregate memory of the Knowledge Cache enough to see the benefit from parallel memory accesses; currently our workloads contain 40 queries, if this was increased and the query selection tuned to generate larger outputs, we would see gains as we added more server nodes. Again, we see benefit from executing a Structured$X$ workload versus Random, with Structured60 and Structured90 showing the highest gains. When a query has a higher probability of being similar to another query in the workload, there becomes a higher probability that a stored knowledge object can match a query with a high reuse score and provide usable information in the knowledge object.
Figure 4.4: Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Structured60 workload.

**Multiple Client, Multiple Servers**

The execution times of the Random, Structured30, Structured60 and Structured90 workloads are shown in figures 4.7, 4.8, 4.9, 4.10 in similar fashion to the preceding section. Again we see performance increases when we introduce the Knowledge Cache across all of the workloads. We see modest gains on several clients in the Random and Structured30 workloads when switching from a configuration where each client uses its own single server node, to one where all four clients share a common server node. When multiple clients share a Knowledge Cache, the knowledge objects that are stored can be matched to any query submitted to the system, thus the degree of sharing of the knowledge objects increases; in this experiment we are seeing some clients benefit
Figure 4.5: Processing time (in seconds) for $k$-Means clustering workloads, four clients, varying server configurations, Structured90 workload.

Figure 4.6: Processing time (in seconds) for frequent itemset mining workloads, single client, varying server nodes.
from retrieving knowledge provided by another client. The gains seen by sharing a

cache appear to disappear in the Structured60 and Structured90 workloads. Although
disappointing, it is not unexpected for the higher probability structured workloads.
The chance of re-use depends very much on the distribution of the queries that are
submitted to the cache. In a Structured90 workload the queries have high chance
of being ‘similar’ to a prior query. If the four clients are processing disjoint queries,
then any knowledge generated will not be useful to another client until the query
parameters overlap and knowledge sharing can begin. If the workloads have a high
chance of remaining disjoint, there is a low chance of being able to share a knowledge
object. Again, because of the limited workload size (40 queries for each query file)
we do not fill the aggregate memory enough to see benefits from increasing the server
node count.

4.1.3 System Overhead

We wish to briefly examine the overhead coupled with using the Knowledge Cache.
For each query in each workload we tracked the overhead associated with the querying,
retrieval, dilinearization, linearization and transmission of knowledge objects to and
from the Knowledge Cache. Summarized in table 4.1.3 are the components of the total
time to process the workload which are spent in each main phase. This data is taken
from the Random workload with one server node used in section 4.1.2. This particular
workload had a cache hit rate of 45%, meaning that 45% of the queries could have
delivered a knowledge object with reuseScore over the threshold. We can see that
the overhead of using the Knowledge Cache is extremely low, with less than 0.06%
of the total processing time being spent processing and transmitting the knowledge
Figure 4.7: Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Random workload.

object. This low overhead can be explained through many of the optimizations and design considerations made when developing the Knowledge Cache. The Knowledge Caching API allows the programmer to provide indexing fields which enabled the server to select candidate knowledge objects quickly, even when the Knowledge Cache fills. Additionally, the server side is build upon the MPI message passing interface, on top of the ultra-high-bandwidth, low latency interconnect infiniband. The transfer of the knowledge objects to the client occurred over 1Gbps on the same network as the server.
Figure 4.8: Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Structured30 workload.

Figure 4.9: Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Structured60 workload.
Figure 4.10: Processing time (in seconds) for frequent itemset mining workloads, four clients, varying server configurations, Structured90 workload.

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Time (seconds)</th>
<th>Overall Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive and Dilinearize</td>
<td>0.91</td>
<td>0.020%</td>
</tr>
<tr>
<td>Mine</td>
<td>4670.75</td>
<td>99.94%</td>
</tr>
<tr>
<td>Linearize and Transmit</td>
<td>1.76</td>
<td>0.038%</td>
</tr>
</tbody>
</table>

Table 4.1: System overhead of a Random query workload.
Chapter 5: CONCLUSION AND FUTURE WORK

In this thesis we presented the Knowledge Cache, a high-performance service for the storage, indexing and retrieval of knowledge. The Knowledge Cache is designed to significantly increase the performance of data mining algorithms and therefore the usability of data mining systems that are deployed for interactive, exploratory data mining. Response time is key to enabling the extraction of useful information from data and every mechanism by which data mining algorithms can be accelerated to complete quicker and deliver answers sooner gets us closer to finding the important facts hidden within data. The Knowledge Cache employs an object based design and uses the abstraction of a knowledge object to encapsulate and represent knowledge, and a query object to encapsulate a request for knowledge. The Knowledge Cache can run as a multi-node server entity leveraging parallel compute and aggregate memory capacity during runtime. The Knowledge Cache is built upon state of the art ultra-high bandwidth interconnects and uses open source tools for simple deployment. The Knowledge Cache dynamically manages load balance by placing data equally across the aggregate memory of the server pool, and manages replacement of knowledge objects when the cache is will. The Knowledge Cache can serve multiple clients concurrently and can share knowledge objects between all clients, and as the specifics of the implementations of the knowledge objects are hidden from the system, the
Knowledge Cache can store a mix of knowledge objects of any type and size enabling use in a heterogeneous environment with varying execution demands.

In addition we presented the Knowledge Caching API, a full user programming interface for the simple integration of the Knowledge Cache into data mining solutions. The API gives the user flexible unobtrusive tools for building knowledge objects and query object. The user has full control over how to represent knowledge in the cache with the single limitation that the knowledge must be linearizable into a contiguous block for storage. Beyond that, the user can elect to store any information inside an object from simple constants to complicated data structures or program state. The Knowledge Caching API’s template also allows the user to programmatically define whether or not a knowledge object is suitable for a particular and define how to rank the appropriateness of a knowledge object for a query.

After presenting the design considerations for the Knowledge Cache and detailed its architecture and implementation, we discuss what it means for an algorithm to be accelerated through the inclusion of knowledge as well as offer guidance as to how one can recognize opportunities for knowledge extraction and acceleration through reuse. Prior knowledge is most effective when there is much redundant computation and data access patterns across algorithm executions. We walked through two start-to-finish examples of how to use the Knowledge Cache API for two classical data mining problems: \( k \)-Means clustering and frequent itemset mining.

Lastly we experimentally evaluated the performance of the system as a whole on the data mining mentioned algorithms above. We tested many configurations of client and server to understand runtime characteristics of the system as well as the benefits of opening up algorithms for knowledge reuse. The Knowledge Cache
can provide significant performance increases and is of considerably low overhead. The evaluation saw execution times improve by up to a factor of 3.18 in the case of $k$-Means clustering and 2.98 for frequent itemset mining.

A unique aspect of the Knowledge Cache is how easily a single client can benefit from the shared data in the cache. Clients contribute knowledge objects to the Knowledge Cache which builds a communal pool of knowledge from which any client can use to their benefit. A new client may connect to the knowledge cache and find that the answer to a long running and involved data mining algorithm is available immediately or after only a short amount of post-processing.

5.1 Limitations of This Thesis

Despite our best efforts, the Knowledge Caching framework detailed in this thesis does possess a number of inherent limitations which we detail in this section. Firstly, in order to utilize the Knowledge Cache the algorithm in question must be amenable to acceleration through re-use; unfortunately there are classes of data mining algorithms which do not meet this criteria. Algorithms such as anomaly detection, or algorithms which are very sensitive to changes in the data set or parameters, will not benefit from the inclusion of knowledge. The Knowledge Cache also uses a naïve cache replacement policy, FIFO. There are arguments for other caching policies, such a least-recently-used, which could yield higher performance due to the retaining of heavily accessed knowledge objects longer. The experimentation did not fill the cache so the caching policy was not tested. Currently, the Knowledge Cache is not persistent, that is there is no way to save the state of the server nodes when the server has exited. When that happens, the cache must be re-filled with results from subsequent
algorithm executions. A further limitation was present within the experimental eval-
uation through the use of quite small query files, 100 queries for $k$-Means clustering
and 40 queries for frequent itemset mining. Because of the small amount of total
knowledge object data produced from these query files, using multiple nodes often
resulted in the intra-server I/O being more expensive that using a single node, which
was demonstrated in some of the experimental results. Finally, although the Knowl-
edge Cache can yield considerable savings when knowledge is extracted and re-used,
there is a burden on the individual to first understand and profile the processing and
data access patterns of an algorithm and furthermore to open up and redesign the
algorithm for the inclusion of knowledge, which in some cases is a non-trivial exer-
cise. With these limitations in mind however, we feel that the investment in time to
re-factor and implement knowledge accelerated algorithms is a worthwhile exercise to
realize the gains in efficiency and response time.

5.2 Future Work

This thesis presents solutions which take an initial step in the direction of optimiz-
ing interactive exploratory data mining through the introduction and the extraction
of knowledge, paving the way for multiple directions for further investigation, while
also observing the current limitations of the existing work. When considering the
system as a whole the Knowledge Cache is currently a straightforward cache which
uses a FIFO replacement policy. Given than knowledge objects of different size can
reside on the Knowledge Cache, the investigation of other cache replacement policies,
such as non-uniform replacement [3], is worthwhile. On the server, the knowledge
objects exist as binary blobs and the only execution performed server side is testing
reuse and reuse score. Directly manipulating the knowledge objects server-side would
be an interesting avenue for exploration. Could a simple server-side modulation of a
knowledge object mean the difference between matching and not matching a query.
Could multiple knowledge objects be combined in certain situations? This thesis did
not examine the cache in a high demand scenario, could the Knowledge Cache learn
what knowledge objects are likely to be requested based on incoming query patterns
and attempt to fill itself to meet the expected demand? When considering the ex-
ample implementation of frequent itemset mining this thesis did not isolate whether
or not a closely matching support value or high over-lap range results in the most
beneficial knowledge objects being matched to queries. Exploration of this factor
would be an interesting extension of this work.
Bibliography


[6] David W. Cheung, Jiawei Han, Vincent T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique, 1996.


[15] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach, 2004.


Appendix A: KNOWLEDGE CACHING API SOURCE CODE

Listing A.1: The Abstract Knowledge Object Class

```cpp
#include "abstractMetaData.hpp"
#include "abstractKnowledge.hpp"
#include "abstractQuery.hpp"
#include <vector.h>
#include "index.h"

/* abstract class for a knowledge object */

class abstractKO
{
    public:

    /* id for knowledge object */
    long id;

    /* pointer to meta data */
    abstractMetaData *ivAbstractMetaData;

    /* pointer to knowledge */
    abstractKnowledge *ivAbstractKnowledge;

    /* method to check if knowledge object is re-usable */
    virtual int canReuse(abstractQuery*) = 0;

    /* method to evaluate the reuseScore for a knowledge object */
    virtual float reuseScore(abstractQuery*) = 0;

    /* method to set meta data */
    virtual void setMetaData(abstractMetaData *md) {ivAbstractMetaData = md;}

    /* method to set knowledge */
```
virtual void setKnowledge(abstractKnowledge *kw) {ivAbstractKnowledge = kw;}

/* method to get meta data */
abstractMetaData * getMetaData() {return ivAbstractMetaData;}

/* method to get knowledge */
abstractKnowledge *getKnowledge() {return ivAbstractKnowledge;}

/* method to get index field types */
/* 0 – integer */
virtual void getIndexFieldTypes(vector <int> *) = 0;

/* method to get index field relationships */
/* 0 ==, 1 <= */
virtual void getIndexFieldRelations(vector <int> *) = 0;

/* method to get index field values */
virtual void getIndexFieldValues(vector <long> *) = 0;

virtual ~abstractKO(){};
}; //End class abstractKO

Listing A.2: The Abstract Knowledge Class

/* abstract class for knowledge */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <vector.h>

class abstractKnowledge
{
    public:

    /* binary image */
    char *bin;
    vector <int> pageids;

    /* size of bin */
    long i_size;

    /* return a binary image of the knowledge */
    virtual void* linearize() = 0;
Listing A.3: The Abstract Meta Data Class

/* abstract class for knowledge object meta data */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

class abstractMetaData
{
public:

    /* pointer binary image when linearized*/
    char *bin;

    /* size of binary image */
    long isize;

    /* return a binary image of the meta data */
    virtual void* linearize() = 0;

    /* return size of binary image of meta data */
    virtual long size() = 0;

    /* reconstruct meta data from binary image */
    virtual void dilinearize(void *, long) = 0;
};
Listing A.4: The Abstract Query Object Class

/* abstract class for knowledge object query */

#include <stdlib.h>
#include <string.h>
#include <vector.h>
#include "index.h"

class abstractQuery
{
    public:

    /* binary image */
    char *bin;

    /* size of binary image */
    long i_size;

    /* return a binary image of the query */
    virtual void *linearize() = 0;

    /* return size of binary image of meta data */
    virtual long size() = 0;

    /* reconstruct query from binary image */
    virtual void dilinearize(void *, long) = 0;

    /* copy query */
    void copy(void *c, long size){bin = (char*) malloc(size); i_size = size; memcpy(bin, c, size);}

    virtual ~abstractQuery() {}

    /* method to get index field types */
    /* 0 integer */
    virtual void getIndexFieldTypes(vector <int> *) = 0;

    /* method to get index field relationships */
/ 0 ==, 1 <= */
virtual void getIndexFieldRelations(vector <int> *) = 0;

/* method to get index field values */
virtual void getIndexFieldValues(vector <long> *) = 0;

}; // End class abstractQuery
Appendix B: EXAMPLE IMPLEMENTATION - K-MEANS CLUSTERING

Listing B.1: The Knowledge Object Class for k-Means clustering

```cpp
#include "abstractKO.hpp"

/* class for a knowledge object */

class kMeansKO: public abstractKO
{
public:

kMeansKO();
virtual int canReuse(abstractQuery* q);
virtual float reuseScore(abstractQuery* q);
virtual void getIndexFieldTypes(vector<int>*);
virtual void getIndexFieldRelations(vector<int>*);
virtual void getIndexFieldValues(vector<long>*);
long hash(char*, long);
~kMeansKO();
}; // End class kMeansKO: public abstractKO
```

Listing B.2: The Knowledge Object Class Definition for k-Means clustering

```cpp
#include "kMeansKO.hpp"
#include "kMeansQuery.hpp"
#include "kMeansMetaData.hpp"
#include <stdio.h>

kMeansKO::kMeansKO(){}

int kMeansKO::canReuse(abstractQuery *q)
{
    /* dinlinearize */
```
if (ivAbstractMetaData->bin)
    dlinearize(ivAbstractMetaData->bin, ivAbstractMetaData->i.size);

/* determine early if the ko is usable */
if( ((kMeansQuery*)q)->ivNumCenters >
    ((kMeansMetaData*)ivAbstractMetaData)->ivNumCenters)
    return 0;

/* determine whether the ko and qo data ranges overlap */
if( !strcmp(((kMeansQuery*)q)->ivdname,
    ((kMeansMetaData*)ivAbstractMetaData)->ivdname) &&
    ((kMeansQuery*)q)->ivRangeStart <=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeStart &&
    ((kMeansQuery*)q)->ivRangeEnd >=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd ) ||
    ((kMeansQuery*)q)->ivRangeStart <=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeStart &&
    ((kMeansQuery*)q)->ivRangeEnd >=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd ) ||
    ((kMeansQuery*)q)->ivRangeStart <=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd &&
    ((kMeansQuery*)q)->ivRangeEnd >=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeStart ) ||
    ((kMeansQuery*)q)->ivRangeStart >=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeStart &&
    ((kMeansQuery*)q)->ivRangeEnd <=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd ) ||
    ((kMeansQuery*)q)->ivRangeStart >=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd &&
    ((kMeansQuery*)q)->ivRangeEnd <=
    ((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd )
{
    return true;
}

/* no overlap — not reusable */
return false;

float kMeansKO::reuseScore(abstractQuery *q)
{
    float queryStart = (float)((kMeansQuery*)q)->ivRangeStart;
    float queryEnd = (float)((kMeansQuery*)q)->ivRangeEnd;
    float metaStart = (float)((kMeansMetaData*)ivAbstractMetaData)->ivRangeStart;
    float metaEnd = (float)((kMeansMetaData*)ivAbstractMetaData)->ivRangeEnd;
    float overlap;
}
/* get the overlap amount */
if(queryStart < metaStart)
    overlap = queryEnd - metaStart;
else
    overlap = metaEnd - queryStart;

float queryCenters = (float)((kMeansQuery*)q)->ivNumCenters;
float metaCenters = (float)((kMeansMetaData*)ivAbstractMetaData)->ivNumCenters;

/* compute range and overlap scores */
float rangeScore = overlap / (queryEnd - queryStart > metaEnd - metaStart ? queryEnd - queryStart : metaEnd - metaStart);
rangeScore *= 100.0;
float centerScore = (queryCenters < metaCenters ? queryCenters : metaCenters) * 100 /
    (queryCenters > metaCenters ? queryCenters : metaCenters);

return rangeScore + centerScore;

kMeansKO::~kMeansKO()
{
    delete(ivAbstractMetaData);
    delete(ivAbstractKnowledge);
}

void kMeansKO::getIndexFieldTypes(vector<int> *types)
{
    /* tell the knowledge cache what types the infex fields are */
    types->push_back(INTE);
}

void kMeansKO::getIndexFieldRelations(vector<int> *relations)
{
    /* for each index type, define the relationship */
    relations->push_back(EQ);
}

void kMeansKO::getIndexFieldValues(vector<long> *values)
{
    /* provide the knowledge cache the values of the index fields */
    if(ivAbstractMetaData -> bin)
ivAbstractMetaData -> dilinearize(ivAbstractMetaData -> bin,
    ivAbstractMetaData -> i_size);
 /* we hash the filename for easier comparison */
 values->push_back(hash( ((kMeansMetaData*)ivAbstractMetaData)->ivdname,
    strlen(((kMeansMetaData*)ivAbstractMetaData)->ivdname)));
}

long kMeansKO::hash(char *bin, long size)
{
    long hash = 0;
    for(long i=0; i < size; i++)
        hash += bin[i];
    return hash;
}

Listing B.3: The Knowledge Class for k-Means clustering

#include <vector.h>

/* ptree knowledge */
class kMeansKnowledge: public abstractKnowledge
{
    public:

    /* constructors */
    kMeansKnowledge();
    kMeansKnowledge(void *, long size);

    /* return a binary image of the knowledge */
    virtual void* linearize();

    /* return size of binary image */
    virtual long size();

    /* reconstruct knowledge from binary image */
    virtual void dilinearize(void *, long);

    /* debug method to dump stuff */
    void dumpKnowledge(void);

    /* load a vector with centers from knolwedge */
    void getCentersAsVector(vector<center> &, int);

    /* destructor */
Listing B.4: The Knowledge Class Definition for \textit{k}-Means clustering

```cpp
#include "abstractKnowledge.hpp"
#include "kMeansKnowledge.hpp"
#include <stdio.h>
#include <string.h>
#include <math.h>

double euclid(center, center);
center merge(center, center);

kMeansKnowledge::kMeansKnowledge()
{
    bin = NULL;
}
kMeansKnowledge::kMeansKnowledge(void *k, long size)
{
    bin = (char *)malloc(size);
    memcpy(bin,k,size);
    i_size = size;
}

void* kMeansKnowledge::linearize()
{
    return bin;
}

void kMeansKnowledge::dilinearize(void *k, long size)
{
    if(bin)
    {
        free(bin);
        bin = 0;
    }
    bin = (char *) malloc (size);
    memcpy(bin,k,size);
    i_size = size;
}

/*
Extracts the knowledge as a vector. If metaNum > queryNum, will merge
*/
```

centers until we have the right quantity
*/

void kMeansKnowledge::getCentersAsVector(vector<
center> & k, int numNeeded)
{
   double * p = (double *)bin;
   int numCenters = (int)p[0];
   int dim = (int)p[1];
   for(int i = 2; i < i_size / sizeof(double); i++){
      center c;
      c.c = (double *)malloc(sizeof(double) * dim);
      c.p = (double *)malloc(sizeof(double) * dim);
      for(int j = 0; j < dim; j++){
         c.c[j] = p[i];
         c.p[j] = c.c[j];
      }
      k.push_back(c);
   }
   if(numCenters > numNeeded){
      while(k.size() > numNeeded){
         int first = -1;
         int second = -1;
         double min = 9999999999999999;
         double dist;
         for(int i = 0; i < k.size(); i++)
            for(int j = i + 1; j < k.size(); j++){
               if(i != j){
                  dist = euclid(k[i], k[j]);
                  if(dist < min){
                     min = dist;
                     first = i;
                     second = j;
                  }
               }
            }
         center c = merge(k[first], k[second]);
         k.erase( k.begin() + second );
         k.erase( k.begin() + first );
         k.push_back(c);
      }
   }
   center merge(center a, center b)
```c
{ center ret;
  ret.c = (double *)malloc(sizeof(double) * a.dim);
  ret.p = (double *)malloc(sizeof(double) * a.dim);
  for(int i = 0; i < a.dim; i++)
  {
    ret.c[i] = (a.c[i] + b.c[i]) / 2;
    ret.p[i] = ret.c[i];
  }
  ret.dim = a.dim;
  return ret;
}

double euclid(center p, center c)
{
  double distance = 0.0;
  for(int i = 0; i < p.dim; i++)
    distance += pow((c.c[i] - p.c[i]), 2);
  distance = sqrt(distance);
  return distance;
}

long kMeansKnowledge::size()
{
  return i_size;
}

kMeansKnowledge::~kMeansKnowledge()
{
  if(bin)
    free(bin);
}
```

Listing B.5: The Meta Data Class for k-Means clustering

/* kMeans meta data object */
class kMeansMetaData: public abstractMetaData
{
  public:

    /* range and number of centers being cached */
    int ivRangeStart;
    int ivRangeEnd;
    int ivNumCenters;

78
Listing B.6: The Meta Data Class Definition for k-Means clustering

```cpp
#include "abstractMetaData.hpp"
#include "kMeansMetaData.hpp"
#include <string.h>
#include <stdio.h>

kMeansMetaData::kMeansMetaData()
{
    bin = NULL;
}

kMeansMetaData::kMeansMetaData(char *dset, int rangeStart, int rangeEnd, int numCenters)
{
    /* initialize */
    bin = NULL;
    ivdname = (char *) malloc(sizeof(char) * (strlen(dset) + 1));
    strcpy(ivdname, dset);
}
```
ivRangeStart = rangeStart;
ivRangeEnd = rangeEnd;
ivNumCenters = numCenters;
}

void kMeansMetaData::linearize()
{
    if(bin) return bin;

    /* create binary image */
    char *ptr;
    i_size = sizeof(int) * 3 + sizeof(char) * (strlen(ivdname) + 1);
    bin = (char *) malloc(i_size);

    /* insert the numerical values */
    memcpy(bin, &ivRangeStart, sizeof(int));
    ptr = bin + sizeof(int);
    memcpy(ptr, &ivRangeEnd, sizeof(int));
    ptr += sizeof(int);
    memcpy(ptr, &ivNumCenters, sizeof(int));
    ptr += sizeof(int);

    /* advance pointer to copy filename */
    ptr = bin;
    for(int i = 0; i < (sizeof(int) * 3); i++) ptr ++;
    memcpy(ptr, ivdname, sizeof(char) * (strlen(ivdname) + 1));
    free(ivdname);

    return bin;
}

void kMeansMetaData::dilinearize(void *t_bin, long size)
{
    /* rebuild structure from binary image */
    i_size = size;
    ivRangeStart = *((int*) t_bin);
    char *ptr = (char*) t_bin + sizeof(int);
    ivRangeEnd = *((int *)ptr);
    ptr += sizeof(int);
    ivNumCenters = *((int *)ptr);
    ptr = (char *)t_bin;
    for(int i = 0; i < (sizeof(int) * 3); i++) ptr ++;

Listing B.7: The Query Object Class for k-Means clustering

/* kMeans knowledge object query */
class kMeansQuery: public abstractQuery
{
public:

    /* range for query */
    int ivRangeStart;
    int ivRangeEnd;

    /* number of centers */
    int ivNumCenters;

    /* data set name */
    char *ivdname;

    /* constructors */
    kMeansQuery();

    /* dataset name, range, centers */
    kMeansQuery(char *, int, int, int);

    /* destructor */
    ~kMeansQuery();

    /* return a binary image of the meta data */
    virtual void* linearize();

    /* return size of binary image of meta data */
    virtual long size();

    /* reconstruct meta data from binary image */
    virtual void dilinearize(void *, long);
    virtual void getIndexFieldTypes(vector <int> *);
    virtual void getIndexFieldRelations(vector <int> *);
    virtual void getIndexFieldValues(vector <long> *);

    long hash(char *, long);
}; //End class kMeansQuery: public abstractQuery

#include "abstractQuery.hpp"
#include "kMeansQuery.hpp"
#include <stdio.h>
kMeansQuery::kMeansQuery(char *dset, int rangeStart, int rangeEnd, int numCenters)
{
    /* initialize */
    bin = NULL;
    ivdname = (char *) malloc(sizeof(char) * (strlen(dset) + 1));
    strcpy(ivdname, dset);
    ivRangeStart = rangeStart;
    ivRangeEnd = rangeEnd;
    ivNumCenters = numCenters;
}
kMeansQuery::kMeansQuery()
{
    bin = NULL;
}

void* kMeansQuery::linearize()
{
    if(bin) return bin;

    /* create binary image */
    i_size = sizeof(int) * 3 + sizeof(char) * (strlen(ivdname) + 1);
    bin = (char *) malloc(i_size);
    memcpy(bin, &ivRangeStart, sizeof(int));
    char *ptr = bin + sizeof(int);
    memcpy(ptr, &ivRangeEnd, sizeof(int));
    ptr += sizeof(int);
    memcpy(ptr, &ivNumCenters, sizeof(int));
    ptr += sizeof(int);
    ptr = bin;
    for(int i = 0; i < (sizeof(int) * 3); i++) ptr ++;
    memcpy(ptr, ivdname, sizeof(char) * (strlen(ivdname) + 1));
    free(ivdname);
    return bin;
}

void kMeansQuery::dilinearize(void *t_bin, long size)
{
    /* reabuild structure from binary image */
    i_size = size;
ivRangeStart = *((int*) t_bin);
char *ptr = (char*) t_bin + sizeof(int);
ivRangeEnd = *((int*)ptr);
ptr += sizeof(int);
ivNumCenters = *((int*)ptr);

ptr = (char*) t_bin;
for (int i = 0; i < (sizeof(int) * 3); i++) ptr++;
ivdname = (char*) malloc(sizeof(char) * (strlen(ptr) + 1));
ivdname = strcpy(ivdname, ptr);
free(bin); bin = 0;
}

long kMeansQuery::size()
{
    return i_size;
}

kMeansQuery::~kMeansQuery()
{
    if(bin)
        free(bin);
    else
        free(ivdname);
}

void kMeansQuery::getIndexFieldTypes(vector<int> *types)
{
    types->push_back(INTE);
}

void kMeansQuery::getIndexFieldRelations(vector<int> *relations)
{
    relations->push_back(EQ);
}

void kMeansQuery::getIndexFieldValues(vector<long> *values)
{
    if(bin)
        dilinearize(bin, i_size);
    values->push_back(hash(ivdname, strlen(ivdname)));
}
long kMeansQuery::hash(char *bin, long size)
{
    long hash = 0;
    for(long i=0; i < size; i++)
        hash += bin[i];
    return hash;
}