PIR: A Domain Specific Language for Multimedia Information Retrieval

Xiaobing Huang and Tian Zhao
Department of Computer Science
University of Wisconsin – Milwaukee
Milwaukee, WI, U.S.A.
{*xiaobing, tzhao*}*@uwm.edu*

Yu Cao
Department of Computer Science
The University of Massachusetts – Lowell
Lowell, MA, U.S.A.
ycao@cs.uml.edu

***Abstract*—*Multimedia Information Retrieval* (MIR) is a problem domain that includes programming tasks such as salient feature extraction, machine learning, indexing, and retrieval. There are a variety of implementations and algorithms for these tasks in different languages and frameworks, which are difficult to compose and reuse due to the interface and language incompatibility. Due to this low reusability, researchers often have to implement their experiments from scratch and the resulting programs cannot be easily adapted to parallel and distributed executions, which is important for handling large data sets.**

In this paper, we present *Pipeline Information Retrieval* (PIR) – a domain specific language for multi-modal feature manipulation. The goal of PIR is to unify the MIR programming tasks by hiding the programming details under a flexible layer of domain specific interface. PIR optimizes the MIR tasks by compiling the DSL programs into pipeline graphs, which can be executed using a variety of strategies (e.g. sequential, parallel, or distributed execution). We evaluated the performance of PIR applications on single machine with multiple cores, local cluster, and Amazon *Elastic Compute Cloud* (EC2) platform. The result shows that the PIR programs can greatly help MIR researchers and developers perform fast prototyping on single machine environment and achieve nice scalability on distributed platforms.

***Keywords*-DSL; pipeline; multimedia information retrieval; parallel programming; Scala**

I. Introduction

Multimedia Information Retrieval (MIR) (Datta, Joshi, Li, & Wang, 2008; Lew, 2012; Lew, Sebe, Djeraba, & Jain, 2006; Yoshitaka & Ichikawa, 1999) refers to the research endeavor that centers on searching knowledge from multimedia data. In the last decades, substantial progress has been made in different area of MIR research, such as multimedia feature extraction (Hu, Xie, Li, Zeng, & Maybank, 2011; Tuytelaars & Mikolajczyk, 2008), learning and semantics (Atrey, Hossain, El Saddik, & Kankanhalli, 2010; Clinchant, Ah-Pine, & Csurka, n.d.; Wang & Hua, 2011), and high performance indexing and query (Moise, Shestakov, Gudmundsson, & Amsaleg, n.d.; Scherp & Mezaris, 2013; Shestakov, Moise, Gudmundsson, & Amsaleg, n.d.; Mohamed & Marchand-Maillet, 2012). As shown by recent surveys (Datta et al., 2008; Lew, 2012; Lew et al., 2006; Yoshitaka & Ichikawa, 1999), since the year 2000, the MIR research efforts have grown tremendously in terms of the number of researchers and practitioners involved, as well as the research papers published.

As a result of substantial progress of MIR research and applications, many related software packages, libraries, and systems have been developed and evaluated using a wide range of multimedia data. Some prominent examples include the GIFT (the GNU Image-Finding Tool) (CVML, 2007), FIRE (the Flexible Image Retrieval Engine)(Deselaers et al., 2010), Caliph & Emir (Lux, 2009), LIRE (Lucene Image Retrieval) (Savvas & Chatzichristofis, 2008), ImageTerrier (Hare, Samangooei, Dupplaw, & Lewis, n.d.), and OpenIMAJ (Open Intelligent Multimedia Analysis toolkit for Java) (Hare, Samangooei, & Dupplaw, n.d.). While significant progress in both MIR research and software development have been made, in practice, we have witnessed that code reuse and system composition for MIR research are still very difficult and the new system developed on top of existing MIR implementation are not optimized for efficiency and cannot be easily adapted for parallelization, which is essential for handling large multimedia data sets. There are often steep learning curves for researchers to understand and appropriately use existing libraries for their needs. Moreover, a lot of components of the MIR software libraries are sequential programs that are designed to run on shared-memory computer architectures. MIR experiments of large data sets are time consuming and resource intensive; they often take hours to days to complete and some may even fail after exhausting main memory.

Recent development of distributed computing platforms such as Cloud-based services provides great opportunity to reduce runtime cost of large MIR experiments. A Cloud-based service such as Amazon EC2 allows users to distribute workload to many worker nodes, thereby reducing the runtime costs for data parallel applications. However, it is not easy to develop MIR applications and deploy them on distributed platforms. An example of this would be to implement a MIR application using a MapReduce framework such as Hadoop and then deploy the application on a distributed computing platform. This presents several challenges to the developers. Firstly, they are often required to have a good understanding of the distributed framework interface they are working with. They need to not only grasp the appropriate usages of different distributed programming artifacts but also understand how to deploy the application on the distributed computing environments that mostly demand platform-specific configurations, settings, and performance tuning. Secondly, They frequently need to adapt the sequential MIR libraries to run in distributed settings where subtle errors such as race conditions can arise. For example, if a user runs some parallel data processing task using several worker nodes while the task program calls a MIR library component that writes to a static variable, a race condition will form if subsequent computation depends on the variable. Lastly, developers often need to write different versions of code (e.g. sequential vs. shared-memory parallel vs. distributed-memory parallel) for the same tasks for performance comparison. The different versions involve a lot of boilerplate code duplication, system configuration, and troubleshooting. In other words, users are forced to spend a large amount of time on non-domain-specific work that greatly reduces research productivity.

To address these problems, we introduce Pipeline Information Retrieval (PIR) language, a DSL for developing MIR applications. PIR is an embedded DSL using Scala (*cf.* http://scala-lang.org/) as the host language: the DSL programs are plain Scala programs and the DSL compiler and runtime are Scala libraries. Users can construct MIR applications with simple DSL constructs that focus more on the high-level logic of the MIR application and less on the implementation details for resource management, optimization, and parallelization. PIR provides an abstraction layer over the concrete implementations of various MIR algorithms for feature extraction, machine learning, indexing, and query. This layer separates the construction of a MIR workflow from its execution so that users can choose optimal execution strategies with minimal changes to the program source. Since PIR is embedded in Scala, it is able to utilize the type system of Scala to ensure that DSL programs are well-typed. That is, if a PIR program compiles in Scala, then it will execute according to the DSL semantics. The execution of a PIR program includes two stages. In the first stage, the PIR program is "compiled" into a pipeline graph through a Scala library. In the second stage, the pipeline graph is executed either sequentially, as a shared-memory parallel program, or on a distributed platform. The construction of the pipeline is separated from its execution to enable flexible execution strategies. For instance, users can use sequential execution to test the correction of an application and use parallel or distributed strategy for better performance. PIR runtime handles the details of deploying the applications on a distributed platform (e.g Amazon EC2) using a MapReduce framework (e.g. Spark (Zaharia, Chowdhury, Franklin, Shenker, & Stoica, 2010)).

In summary, this paper presents the design and implementation of a DSL for writing MIR programs using simple language constructs. This DSL abstracts away the implementation details of MIR algorithms to enable rapid prototyping of MIR applications and allows users to apply various runtime strategies with minimal change to the program source. Users of the DSL can focus more on the application logic of MIR programs and resulting programs are more readable and reusable.

In the rest of the paper, we first explain the background of multimedia retrieval and the motivation of PIR language in Section II. We then give an informal presentation of PIR language in Section III using examples. The formal syntax and semantics of the language are covered in Section IV, V, and VI. We include some details of our implementation in Section VII and the results of our experiments in Section VIII. The related works are in Section IX.

II. Multimedia Information Retrieval

Due to the complication of MIR applications, it is often the case that MIR applications utilize multiple libraries to implement their workflows.

A typical MIR workflow starts with extracting salient features from multimedia objects such as text documents, images, and videos. After the feature extraction, some MIR algorithms may require applying machine learning algorithms to discover latent semantics within the features and use the results to transform the extracted features. MIR workflows for large data sets may also index the extracted features for better performance. For evaluation, many MIR workflows include query tasks to retrieve multimedia objects similar to query objects from indices or feature sets.

For example, the LIRE library is a *Content-Based Image Retrieval* (CBIR) library that implements multiple image analysis algorithms to extract global features based on color, edge, and texture, and local features such as *Scale-Invariant Feature Transform* (SIFT) (Lowe, 2004), *Speeded Up Robust Feature* (SURF) (Bay, Tuytelaars, & Van Gool, 2006), and *Maximally Stable Extremal Regions* (MSER) (Matas, Chum, Urban, & Pajdla, 2004). Image retrieval can be based on the similarity of image features. It can also be based on the bag-of-visual-word approach, where features of training images are clustered into centroids (visual words) and the features of images are converted to histograms of visual words. LIRE uses Lucene library to index the image features. LIRE's indexing implementation is rather simple and it translates image features or visual word histograms into Lucene documents for indexing in Lucene engine, which can be inefficient. Thus, if better performance is needed for indexing multimedia data, Libraries such as ImageTerrier should be used. Also, LIRE provides limited support for machine learning such as K-Means (Hartigan & Wong, 1979) and *Latent Semantic Analysis* (LSA) (Dumais, 2004). If more sophiscated machine learning algorithms, e.g. as *Latent Dirichlet Allocation* (LDA) (Blei, Ng, & Jordan, 2003), are needed, then other libraries such as MALLET (McCallum & Kachites, 2002) must be included as well.

Using multiple libraries in a MIR application complicates development effort. Each MIR library has its own API for MIR data representation. A MIR application that uses multiple libraries needs to include glue code to convert of results from one library to another. Also, each library has its own API for MIR tasks such as feature extraction and machine learning. Mixing these APIs in an application obscures the main logic of the application, which makes the application difficult to understand and less reusable. In addition, most MIR libraries are sequential programs. Mixing libraries directly in a MIR application increases the workload for implementing parallel or distributed solutions. A lot of development time is wasted on adapting sequential programs to parallel execution and on deployment to distributed computing platforms.

III. Pipeline Information Retrieval Language

PIR is designed to address the above problems by providing language-based abstractions for multimedia data representation and manipulation. PIR serves the purpose of glue code that allows users to access functionalities in multiple MIR libraries without knowing the specifics of the libraries. PIR also provides runtime support that can switch between different execution strategies without modifying the source program.

PIR provides API functions for invoking the implementations of MIR tasks in libraries. Examples of the API functions are listed in Figure 1. Users can construct MIR workflows using high-level PIR constructs and the API functions. The API functions implement common interfaces that represent common MIR tasks that include data loading, feature extraction (transformation), machine learning, indexing, and querying. Users can also implement their own API functions for new MIR tasks or new libraries. PIR provides operators to assemble these API functions into pipeline-based workflows.

A. Examples

We demonstrate the usage of PIR with examples written in a simplified syntax (detailed in Section IV). The actual programs include slightly more details such as parameters.

Listing 1 is an example for image retrieval using two global image features: *Color and Edge Directivity Descriptor* (CEDD) (Chatzichristofis & Boutalis, 2008a) and *Fuzzy Color and Texture Histogram* (FCTH) (Chatzichristofis & Boutalis, 2008b). This example uses PIR operators and API functions for MIR operations. For example, `load("index_image")` loads images from the file path `index_image`. In `img.connect(f_cedd)`, the loaded images are connect to a projection function `f_cedd`, which extracts CEDD features from images. Also, `index` operator creates a Lucene index using `f_luceneIdx` and the extracted features, while `query` operator applies the function `f_weightedQuery` to the index and query features.

API functions	MIR functionality
f_cedd	CEDD feature extraction
f_fcth	FCTH feature extraction
f_gabor	Gabor feature extraction
f_colorLayout	color layout feature extraction
f_sift	SIFT feature extraction
f_ldaProj	LDA topic histogram
f_cluster	clustering histogram
f_transmedia	transmedia projection
f_ldaTrain	LDA training
f_kMeansTrain	K-Means clustering
f_ccaTrain	CCA training
f_luceneIdx	Lucene indexing
f_weightedQuery	joint query with weights
f_distance	distance between features

Figure 1: Examples of API functions in PIR

Listing 1: Image Query Example

```
val img = load("index_image")
val qImg = load("query_image")

val idx = index(f_luceneIdx,
                img.connect(f_cedd),
                img.connect(f_fcth))
val q =  query(f_weightedQuery, idx,
                qImg.connect(f_cedd),
                qImg.connect(f_fcth))
q.collect
```

The first 4 statements in Listing 1 construct a pipeline graph with each vertex of the graph corresponds to an operation such as image loading, feature extraction, indexing, and querying. The actual computation is delayed until the last statement `q.collect`, which executes the pipeline graph using parallel threads to answer the query `q`. Delayed execution allows the DSL runtime to manage how to execute the function of each pipeline vertex and how the intermediate results of the vertex are stored and reused. In this example, the image loading, feature extraction, and indexing vertices are run in parallel. The caching of the intermediate results is automatic so that they can be stored in memory if they fit, otherwise file system will be used for storage purpose.

The API functions such as `f_cedd`, `f_fcth`, and `f_luceneIdx` are thin wrappers for the implementations adapted from the MIR libraries. Users can expand the capability of the DSL by defining similar functions with appropriate type signatures.

The second example (Listing 2) illustrates a transmedia multi-modal retrieval experiment that uses a `train` operator to apply training algorithms such as LDA, K-Means, and *Canonical Correlation Analysis* (CCA) (Thompson, 2005) algorithm to text and image data. The variable `lModel` corresponds to a pipeline that trains a LDA topic distribution model from text using the training function f_ldaTrain. The variable kModel corresponds to a pipeline that applies K-Means clustering algorithm using the function f_kMeansTrain to siftImg, which is a pipeline for extracting SIFT features from images. The CCA model (line 9) is trained on topic distributions of text (line 10) and clustered image features (line 11). Note that txt.connect(f_ldaPrj, lModel) is a pipeline that converts text into LDA topic distribution histogram based on the LDA topic model obtained from lModel. This example applies query image to the trained CCA model to obtain the related text documents.

PIR allows the definition of reusable pipelines. For example, the variable f is a reusable pipeline that extracts SIFT features from images, transforms the features to a histogram of K-Means centroids, and then uses a CCA model to obtain related text files. The variable p is a pipeline that loads a query image and then connects to f to obtain the intended text files. Similar to the first example, the actual computation does not start until the last statement p.collect.

Listing 2: Transmedia Query Example

```
1   val txt = load("training_text")
2   val img = load("training_image")
3   val qImg = load("query_image")
4
5   val siftImg = img.connect(f_sift)
6   val lModel = train(f_ldaTrain, txt)
7   val kModel = train(f_kMeansTrain, siftImg)
8
9   val ccaModel = train(f_ccaTrain,
10        txt.connect(f_ldaPrj, lModel),
11        siftImg.connect(f_cluster, kModel))
12
13  val f = f_sift.connect(f_cluster, kModel)
14            .connect(f_transmedia, ccaModel)
15
16  val p = qImg.connect(f)
17  p.collect
```

We can make changes to the pipeline graph and then run the query again. For example, the following statements changes the file path for the text files and then run the projection p again.

```
txt.f = "training_text2"
p.collect
```

PIR's runtime semantics detectes change propagation such that after the second call to p.collect, the intermediate results of pipeline vertices that depend on txt, such as lModel, ccaModel, and p, are recomputed while the results of other vertices, such as siftImg and kModel, are reused.

The last example (Listing 3) implements the *Series Feature Aggregation* (SFA) algorithm (Zhang & Ye, 2010), which involves sorting and filtering of images based on image features. The SFA algorithm applies multiple filters to a set of images in sequence, where each filter is based on the similarity of visual features to a query image. The irrelevant images are removed after applying each filter and the remaining images are collectively described as similar to the query image by all visual features.

In Listing 3, the variables colorDist, ceddDist, and gaborDist are composite pipelines that compute the distances of images to a query image based on *Color Layout Descriptor* (CLD) (Manjunath, Ohm, Vasudevan, & Yamada, 2001), CEDD, and Gabor (Manjunath et al., 2001) features respectively. The variable img2 is the pipeline that connects the loaded images img1 to a filter function, which is the result of applying a high-order function f_top to a list of image IDs. The argument to f_top is an expression (line 10–12) that computes the distance of input images to the query image based on color distribution features, sorts the image IDs based on distances in ascending order, and keeps the top 2000 image IDs through the call take(2000). Note that we need to invoke collect on img1.connect(colorDist).sort("ascending") to trigger its computation so that we can pass the actual list of image IDs to the function f_top. The function f_top returns a boolean function that evaluates to true for an image argument if the image's ID is among the list of IDs passed to f_top. This function is not a PIR language primitive so that its argument should be an actual value instead of delayed computation (i.e. a pipeline).

The variables img3 and img4 are pipelines that filter the results of img2 and img3 respectively using CEDD and Gabor feature distances. The final results are obtained after the call img4.collect.

Listing 3: Series Feature Analysis Example

```
1 val img1 = load("images")
2 val colorDist = f_colorLayout.connect(
3           f_distance("q_img", f_colorLayout))
4 val ceddDist = f_cedd.connect(
5           f_distance("q_img", f_cedd))
6 val gaborDist = f_gabor.connect(
7           f_distance("q_img", f_gabor))
8
9 val img2 = img1.filter(f_top(
10                img1.connect(colorDist)
11                    .sort("ascending")
12                    .collect.take(2000)
13          ))
14 val img3 = img2.filter(f_top(
15                img2.connect(ceddDist)
16                    .sort("ascending")
17                    .collect.take(500)
18          ))
19 val img4 = img3.filter(f_top(
20                img3.connect(gaborDist)
21                    .sort("ascending")
22                    .collect.take(100)
23          ))
24 img4.collect
```

Next we explain the formal syntax and semantics.

s	$::=$		Statement
		val $x = e$	assignment
	$\mid$	$x.f = f$	update
	$\mid$	$s; s'$	sequence
	$\mid$	$e.collect$	execution
e	$::=$	$src \mid drain \mid p$	Expression
src	$::=$	$load(f)$	load source
	$\mid$	$src.connect(p)$	source pipe
	$\mid$	$src.sort(f)$	sort
	$\mid$	$src.filter(f)$	filter
p	$::=$	$proj(f)$	projection
	$\mid$	$proj(f, e)$	projection with model
	$\mid$	$p.connect(p')$	projection pipe
$drain$	$::=$	$train(f, src)$	model
	$\mid$	$index(f, src)$	index
	$\mid$	$query(f, e_i, e_q)$	query
f			functions and parameters

Figure 2: Pipeline DSL Language Syntax

IV. SYNTAX

The formal DSL syntax is described in Figure 2, where a program consists of a sequence of statements and each statement is a variable declaration, and an update, or an execution. Each variable declaration associates an expression with a local variable. An update statement replaces the file path of a file loading operation. The execution statement in $s;\ e.collect$ triggers the execution of the pipeline graph compiled from s and e and outputs the results of the vertex compiled from e. The expressions include source, drain, and projection (denoted by meta variable p) expressions. A source expression is either a load or a a source pipe expression. A load expression loads the raw data files from a file path and we use a function f to represent the load operation with path to the data file embedded in f. A source pipe expression is a composite of a source expression with a projection expression that transforms the data from the source to a different form. For example, we can implement the SIFT feature extraction from images using a source pipe with a load expression to provide the raw image files and a SIFT extraction function as the projection.

A projection expression is either a direct projection (that may require a model) or a composite projection expression (projection pipe). The projection expression transforms input data using a function and it may take a model expression as a second parameter. The projection expressions represent the extraction of features such as global/local features of images and term frequency histograms of text documents, The second type of pipe expression is for extracting features such as the LDA topic distribution, which requires a LDA topic model obtained from training. A projection pipe can be used for composition of several transformations.

The drain expressions include model, index, and query. The model expression represents the training of a model. Index and query expressions represent the index and query operations. Each of the drain expression takes a function as the parameter. The model expression uses its second parameter as training data. The index expression creates an index for its second parameter. The query expression takes an index e_i and a query input e_q to compare against the index.

Note that though not shown, the concrete syntax we have implemented allows more general cases such as $index(f, e_1, \ldots, e_n)$ to index several types of data. Also note that with the use of implicit conversion functions in Scala, we can omit $proj$. For example, in the table below, the expressions on the left are implicitly converted to expressions on the right.

PIR expression	Result of implicit conversion
`img.connect(f_sift)`	`img.connect(proj(f_sift))`
`txt.connect(f, model)`	`txt.connect(proj(f, model))`
`f_sift.connect( f_cluster, kModel)`	`proj(f_sift).connect( proj(f_cluster, kModel)`

V. FROM DSL TO PIPELINE GRAPH

In this section, we describe the pipeline graph that a DSL program is compiled to and the compilation process. For brevity, we omit the semantics for filtering and sorting, which are treated as special cases of the projection operations.

A. Pipeline Graph

In order to execute the DSL program using flexible runtime strategy, we first compile the DSL program to a pipeline graph with vertices connected through their fields. Each vertex is an instance of the type V as described below.

v	$\in$	V	Vertex
V	$::=$	$V_l(f)$	load source
	$\mid$	$V_{sp}(s, p)$	source pipe
	$\mid$	$V_{pp}(l, r)$	projection pipe
	$\mid$	$V_p(f)$	projection
	$\mid$	$V_{pm}(f, t)$	projection with model
	$\mid$	$V_i(f, s)$	indexing
	$\mid$	$V_t(f, s)$	training
	$\mid$	$V_q(f, i, q)$	query

The fields of each vertex type are shown above. Specifically, the load-source vertex type $V_l(f)$ has a field f to store the function (and file path) for loading source files. A source vertex is either V_l or V_{sp}. A projection vertex can be V_p, V_{pm}, or V_{pp}. $V_{sp}(s, p)$ is a source pipe with s stores the incoming source vertex and p stores the projection vertex. $V_{pp}(l, r)$ is a projection pipe with l and r being the left and the right projection vertex respectively. $V_i(f, s)$ stores the indexing function in f field and source vertex in s field. $V_t(f, s)$ stores the training function in f field. $V_q(f, i, q)$

stores the querying function in f field, index vertex in i, and query vertex in q. Each vertex except that of projection types uses a field $data$ to store the intermediate or final results computed at the vertex. The `data` field of a load-source vertex is reset to null if its field f is updated.

B. Compilation

We define a denotational semantics as shown in Figure 3 to describe the compilation of a DSL program into a pipeline graph. The semantics includes compilation rules for each type of expression e and statement s. Each expression rule in Figure 3 is written as $[\![e]\!]\sigma = v$ and each statement rule is written as $[\![s]\!]\sigma = \sigma'$. The expression rule states how an expression is reduced to a vertex given a runtime state σ that maps variables to vertices. The statement rule states how a statement changes a state to another. In the rules, a new vertex is created in $v = \mathsf{new}\ V(f)$, where V is the type of the vertex v and f is the function assigned to $v.f$. In the constructor of V, we initialize $v.\mathtt{data}$ to null.

The compilation of a program s is written as $[\![s]\!]\emptyset = \sigma$ that transforms an empty initial state to σ. To see how an execution statement $e.collect$ triggers actual computation, notice that the antecedent of Rule (Exec), $\textsc{Run}(v)$, is a call that starts the execution of the vertex v reduced from e compiled from the previous statements and e. The call of the form $\textsc{Run}(v)$ can be implemented using various strategies explained in the next section.

Expression constructs

$$[\![\_]\!] : Exp \to (State \to Vertex)$$

$$[\![x]\!]\sigma = \sigma(x) \qquad [\![load(f)]\!]\sigma = \mathsf{new}\ V_l(f)$$

$$[\![proj(f)]\!]\sigma = \mathsf{new}\ V_p(f) \qquad [\![proj(f,e)]\!]\sigma = \mathsf{new}\ V_{pm}(f, [\![e]\!]\sigma)$$

$$[\![train(f,e)]\!]\sigma = \mathsf{new}\ V_t(f, [\![e]\!]\sigma)$$

$$[\![index(f,e)]\!]\sigma = \mathsf{new}\ V_i(f, [\![e]\!]\sigma)$$

$$\frac{[\![e_i]\!]\sigma = v_i \qquad [\![e_q]\!]\sigma = v_q}{[\![query(f,e_i,e_q)]\!]\sigma = \mathsf{new}\ V_q(f, v_i, v_q)}$$

$$\frac{[\![e]\!]\sigma = v_s \qquad v_s \in V_l \mid V_{sp} \qquad [\![e']\!]\sigma = v_p}{[\![e.connect(e')]\!]\sigma = \mathsf{new}\ V_{sp}(v_s, v_p)}$$

$$\frac{[\![e]\!]\sigma = v_p \qquad v_p \in V_p \mid V_{pm} \qquad [\![e']\!]\sigma = v'_p}{[\![e.connect(e')]\!]\sigma = \mathsf{new}\ V_{pp}(v_p, v'_p)}$$

Statement constructs

$$[\![\_]\!] : Stmt \to (State \to State)$$

$$\frac{[\![e]\!]\sigma = v}{[\![x = e]\!]\sigma = \sigma[x \mapsto v]} \quad \text{Asgn}$$

$$[\![x.f = f']\!]\sigma = \sigma[x \mapsto \sigma(x)[\mathtt{data} \mapsto \mathtt{null}, f \mapsto f']] \quad \text{Upd}$$

$$\frac{[\![s]\!]\sigma = \sigma' \qquad [\![s']\!]\sigma' = \sigma''}{[\![s;s']\!]\sigma = \sigma''} \quad \text{Seq}$$

$$\frac{[\![e]\!]\sigma = v \qquad \textsc{Run}(v)}{[\![e.collect]\!]\sigma = \sigma} \quad \text{Exec}$$

Figure 3: Compilation of DSL program to pipeline graph

VI. Execution of the pipeline graph

In this section, we describe the strategies for executing a pipeline graph: $\textsc{Run}(v)$. We first describe a sequential strategy.

A. Seqential execution with caching

The execution results of each vertex are implicitly cached so that $\textsc{Run}^*(v)$ in Figure 4 checks whether we should run the computation at vertex v via the call $run?(v)$ before proceeding with the actual computation $\textsc{Run}(v)$. The call $run?(v)$ returns true either when the data cache of v (if exists) is null or when computation is required for the vertices that v depends on.

For simplicity, we use the notation v_x to represent vertex of the type V_x for all subscript x. For example, v_l represents a vertex of type V_l. The execution of the pipeline graph is demand driven. We don't execute the computation of a vertex if it is not triggered by an execution statement in the DSL program. For each of the vertex type, we execute its computation by applying its embedded operation f via calling its `app` method. If f requires inputs from upstream vertices, we execute these vertices first and then pass their data to f.

For example, the execution of a source pipe vertex v_{sp} triggers the execution of its source vertex $\textsc{Run}^*(v_{sp}.s)$ and its projection vertex $\textsc{Run}^*(v_{sp}.p)$, which may use their cached results. Afterwards, we apply the projection $v_{sp}.p$ to the data of $v_{sp}.s$ via calling $v_{sp}.p.\widehat{\mathtt{app}}(v_{sp}.s.\mathtt{data})$. As shown in Figure 4, the $\widehat{\mathtt{app}}$ method has different behavior depending on the type of $v_{sp}.p$.

B. Data parallel execution

PIR programs can take advantage of parallel and distributed computing architectures such as Cloud services for better performance. To support this, we modify the previous runtime strategy to support data parallel execution of the pipeline graph. The projection operations in our pipeline graph are natural candidates for data parallel execution since the source data can be divided into several portions with each portion being transformed independently.

To execute $\textsc{Run}(v)$ in parallel, we modify the operation $v_{sp}.\mathtt{data} \leftarrow v_{sp}.p.\widehat{\mathtt{app}}(v_{sp}.s.\mathtt{data})$ as follows:

1) Divide $v_{sp}.s.\mathtt{data}$ into n segments.
2) Spawn k worker threads and put them in an available worker pool, where $k \le n$.

Runtime semantics of vertices.

$$\textsc{Run}^*(v) = \begin{cases} \textsc{Run}(v) & \text{if } \mathit{run?}(v) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\textsc{Run}(v_l) = \{v_l.\texttt{data} \leftarrow v_l.f.\mathsf{app}()\}$$

$$\textsc{Run}(v_t) = \{\textsc{Run}^*(v_t.s);\ v_t.\texttt{data} \leftarrow v_t.f.\mathsf{app}(v_t.s.\texttt{data})\}$$

$$\textsc{Run}(v_i) = \{\textsc{Run}^*(v_i.s);\ v_i.\texttt{data} \leftarrow v_i.f.\mathsf{app}(v_i.s.\texttt{data})\}$$

$$\textsc{Run}(v_q) = \{\textsc{Run}^*(v_q.i);\ \textsc{Run}^*(v_q.q);\ v_q.\texttt{data} \leftarrow v_q.f.\mathsf{app}(v_q.i.\texttt{data}, v_q.q.\texttt{data})\}$$

$$\textsc{Run}(v_{pp}) = \{\textsc{Run}^*(v_{pp}.l);\ \textsc{Run}^*(v_{pp}.r)\}$$

$$\textsc{Run}(v_p) = \emptyset \qquad \textsc{Run}(v_{pm}) = \{\textsc{Run}^*(v_{pm}.t)\}$$

$$\textsc{Run}(v_{sp}) = \{\textsc{Run}^*(v_{sp}.s);\ \textsc{Run}^*(v_{sp}.p);\ v_{sp}.\texttt{data} \leftarrow v_{sp}.p.\widehat{\mathsf{app}}(v_{sp}.s.\texttt{data})\}$$

Test whether to run a vertex.

$$\mathit{run?}(v) = (v.\texttt{data} == \texttt{null})\ \vee\ \exists v' \in \mathit{depd}(v).\ \mathit{run?}(v')$$

$$\mathit{depd}(v_{pm}) = \{v_{pm}.t\}$$

$$\mathit{depd}(v_{pp}) = \{v_{pp}.l,\ v_{pp}.r\}$$

$$\mathit{depd}(v_i) = \{v_i.s\}$$

$$\mathit{depd}(v_t) = \{v_t.s\}$$

$$\mathit{depd}(v_q) = \{v_q.i, v_q.q\}$$

Apply the operations in projection vertices

$$v_p.\widehat{\mathsf{app}}(d) = v_p.f.\mathsf{app}(d)$$

$$v_{pm}.\widehat{\mathsf{app}}(d) = v_{pm}.f.\mathsf{app}(d,\ v_{pm}.t.\texttt{data})$$

$$v_{pp}.\widehat{\mathsf{app}}(d) = v_{pp}.r.\mathsf{app}(v_{pp}.l.\mathsf{app}(d))$$

Figure 4: Execution strategy with caching

3) For each unprocessed data segment d, take a worker w from the pool and start processing task for d using w. After the task is completed, save the data in the local cache and return w to the pool.
4) Repeat previous step until the data segments have all been processed.

If we use a high-level distributed framework such as MapReduce, the actual management of the worker threads will be delegated to the framework implementation such as Hadoop or Spark.

VII. Implementation details

In general, we can choose any host language to embed PIR. Practically, the Scala language is a good choice as it provides advanced features, such as lazy evaluation, implicit conversion, closures, mixins, and pattern matching, which simplify PIR's design and Scala is flexible enough for us to plug in and plug out artifacts as we need. For instance, we create vertices (or nodes) in different stages (source, pipe, and drain) with the help of implicit conversion. Moreover, Scala is 100% bytecode compatible with Java code, which allows us to easily reuse a large number of the existing MIR Java libraries. In our experiments, we used the LIRE image retrieval framework to extract global and local image features, MALLET for LDA modeling, Apache Lucene [1] for indexing and query, and a Java library for CCA for transmedia query in experiments in Section VIII-A. Below we discuss two interesting aspects of our preliminary implementation [2].

A. Implicit conversion

To illustrate the use of implicit conversion, consider the expression `img.connect(f_cedd)`, where `img` is a pipeline vertex that loads source images. The `connect` method expects a projection stage argument while `f_cedd` is just a projection function. In this case, the Scala compiler is able to use the implicit function shown in Listing 4 to automatically convert `f_cedd` to a projection stage object.

Listing 4: Implicit Conversion

```
implicit def projToProjStage[In <: IFeature,
                             Out <: IFeature]
(proj: GenericProj[In, Out])
                            = new ProjStage(proj)
```

This conversion is also applied to `f_sift` in `f_sift.connect(f_cluster, kModel)` since the method `connect` is defined in `ProjStage` not in `f_sift`. A similar conversion function converts the argument `(f_cluster, kModel)` to a projection stage object as well.

B. Internal stage object creation and access mechanism

The mechanism for PIR internal stage object creation and access is accomplished via the visitor pattern (VanDrunen & Palsberg, 2004). As illustrated in Figure 5, there are three main Scala traits, SourceComponent, ProjComponent, and TrainComponent, that all inherit the Vertex trait. (A Scala trait is a reusable unit of code that supports inheritance and mixin-class compositions.) Most of the stage classes, except IndexStage and QueryStage, directly inherit from the three main traits. Each stage class represents a stage node in the pipeline graph. Both SourceComponent and ProjComponent have the "connect" methods, which are the basic building tool for the pipeline. The "connect" method in SourceComponent creates a SourceStage node while the "connect" method in ProjComponent connects two pipelines together. The TrainComponent generates a model from any machine learning tasks during the MIR execution and the result is stored in its cache "cacheModel". The Vertex

[1]http://lucene.apache.org

[2]https://github.com/pir-dsl/pir

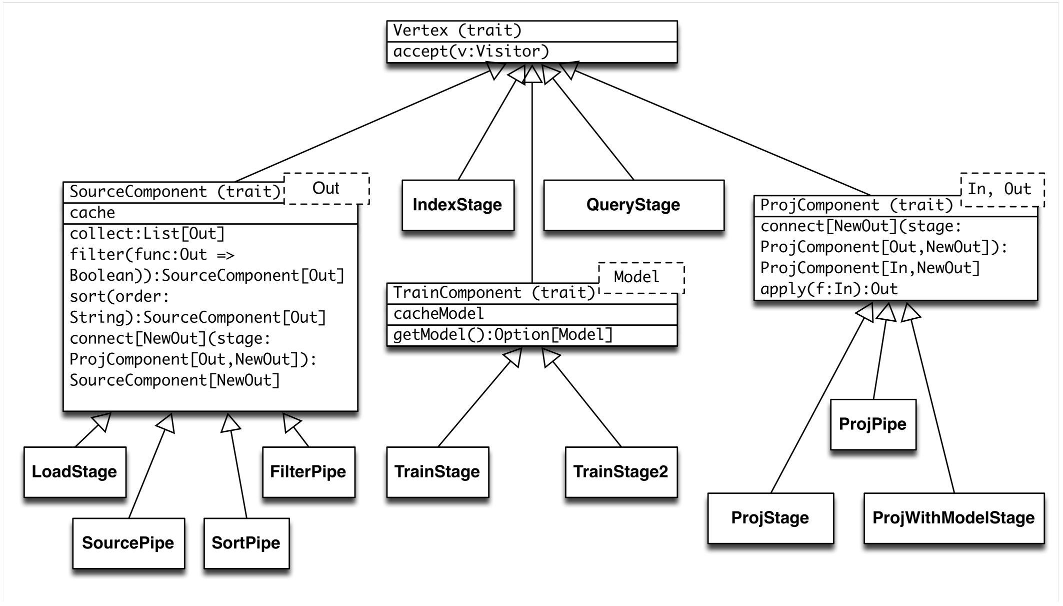

Figure 5: PIR Internal Object Creation and Access Mechanism

trait provides the essential "accept" method. This method takes a visitor as a parameter with type signature shown in Figure 6. We defined a basic RunStrategy trait that is itself a visitor. By extending the RunStrategy trait, we can implement different code execution strategies. Specifically, when the pipeline graph is executed, each stage node will be visited and the result will be stored in its cache and the strategy will decide how the stage node is visited. For example, the image loading will be executed in parallel on all the worker nodes on Amazon EC2 cluster if SparkStrategy is used to visit the LoadStage node.

Finally, we have a JobVisitor class that inherits the Visitor trait as well. Its purpose is to generate a pipeline graph without even running the code. Specifically, with some check-pointing hooks built into each stage node, we can use JobVisitor to draw the pipeline graph through a graph traversal. This is useful to provide PIR users the "big picture" of a complex MIR program so that the correctness of the program from at least the design perspective can be verified before the expensive execution on the actual cluster/cloud is scheduled.

C. Distributed execution

The PIR "compiler" translates the DSL program to a pipeline graph and the PIR runtime can execute the pipeline graph sequentially, in parallel, or on distributed platforms. The parallel execution of PIR programs is straightforward. We implemented a ParallelStrategy class that overrides the visit methods for projection vertices to run projection tasks in parallel threads.

Distributed execution, however, requires more intricate design. Next, we explain the deployment of PIR programs on

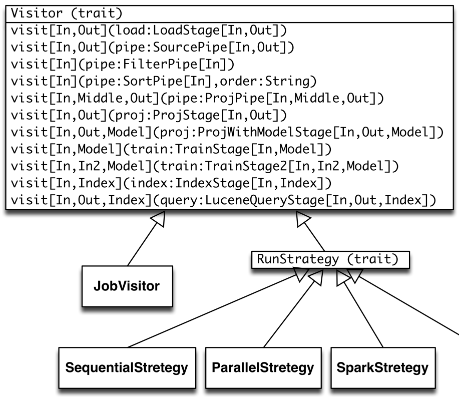
Visitor (trait)
visit[In,Out](load:LoadStage[In,Out])
visit[In,Out](pipe:SourcePipe[In,Out])
visit[In](pipe:FilterPipe[In])
visit[In](pipe:SortPipe[In],order:String)
visit[In,Middle,Out](pipe:ProjPipe[In,Middle,Out])
visit[In,Out](proj:ProjStage[In,Out])
visit[In,Out,Model](proj:ProjWithModelStage[In,Out,Model])
visit[In,Model](train:TrainStage[In,Model])
visit[In,In2,Model](train:TrainStage2[In,In2,Model])
visit[In,Index](index:IndexStage[In,Index])
visit[In,Out,Index](query:LuceneQueryStage[In,Out,Index])
JobVisitor
RunStrategy (trait)
SequentialStretegy
ParallelStretegy
SparkStretegy
......

Figure 6: PIR Stage Node Access via Visitor

Amazon EC2 (Amazon, 2013) using a MapReduce framework – Spark (Zaharia et al., 2010). The system diagram of PIR integration with Spark and Amazon EC2 is illustrated in Figure 7.

We use Spark to provide the distributed runtime environment for its simplicity. In fact, we only need to insert a few lines of source code to PIR runtime to interface with Spark. Spark is a MapReduce framework with memory-based data distribution using Resilient Distributed Datasets (RDD) that can run 10 times faster than Hadoop (Shestakov et al., n.d.). For our experiments, large amount of intermediate data was kept in memory when possible to reduce I/O overhead. For fault tolerance, Spark maintains the lineage of all actions on the RDDs rather than the data itself. If some intermediate data is lost or corrupted, it can be reconstructed by applying

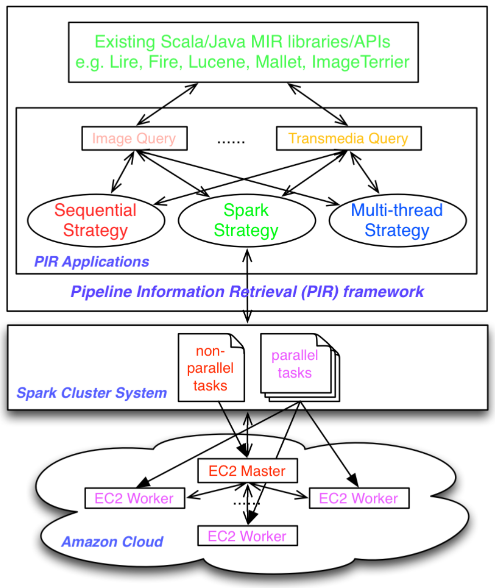

Figure 7: PIR Cloud System Diagram

the lineage chain of actions to the raw data. Through Spark, we can deploy PIR programs on any computing clusters but the most interesting deployment targets are cloud computing platforms since they provide elastic support that can fulfill the needs of big data analytics. For experiments, we selected Amazon EC2 as our deployment target.

D. *Deployment PIR Programs on Amazon EC2*

Spark provides Python scripts for deployment on Amazon EC2 and it also bundles the tool *Ganglia* to monitor the details of PIR execution, such as memory consumption on a particular worker node. For successful deployment of PIR programs on Amazon cloud, we need carefully tuned environment parameters, correctly designed interfaces, and accurately measured results.

1) Deployment of PIR programs on AWS: PIR programs have both parallelizable and sequential components. PIR runtime only passes the parallelizable components to the Spark framework via the MapReduce operations while the sequential components (e.g. KMean cluster) are executed on the master node.

We assemble the PIR program and all related artifacts into a single jar file with the Scala build tool *sbt* so that Spark can send the code from the master node to all worker nodes at runtime for execution. To configure the Spark context initialization parameters, we externalized these parameters for PIR programs to property files, which can be passed to JVM at runtime. For properties that all nodes use, we initially set them in the Spark configuration files on the master node and replicated these files to all worker nodes on the cluster. Spark splits the data into slices before they are distributed to worker nodes, we have made the number of slices a configurable parameter, observing that performance improves with a higher number (e.g. 160) of slices than the default 10. In general, the number of slices should increase with the number of worker nodes.

Spark uses *akka* (a concurrency toolkit for Scala) for concurrent execution. This requires communication between the master and worker nodes or between the application and worker nodes (e.g. passing results from worker nodes to the application). We adjusted *akka* frame size parameter to 150 - 250 MB to properly accommodate the size of the intermediate results. In PIR's distributed computation, the intermediate data are kept in memory using the Spark's persist function, which improves the performance significantly.

2) Configuration of AWS instances: Amazon divides its cloud resources by zones and allocates available resources dynamically. Since Spark needs to initialize the cluster context with the master node's host domain name or IP, it is a good practice to associate an Elastic IP to the master node's dynamic domain name so that the same IP address can be reused when the EC2 instances are restarted.

We initially performed the scalability tests by incrementally adding more worker nodes to the cluster. However, due to Amazon's dynamic EC2 resources allocation, every new test will likely run on different set of cluster nodes and the performance may be significantly skewed. Thus, we requested the maximum number of nodes that we use (e.g. 20 nodes) and then dropped 2 nodes at a time for each subsequent run to reduce the impact of cluster node variation between test runs.

With dynamic EC2 allocation, it is not practical to deploy the raw data to EC2 each time we start a new experiment. Hence we stored all raw data on the Amazon S3 storage service. For this purpose, we designed a simple interface for interaction between the PIR runtime and Amazon S3 storage.

3) Job scheduling of AWS instances: We set the Spark scheduler mode to *FAIR* so that available resources are assigned to jobs in a "round robin" fashion. That is, all Spark jobs received roughly equal share of cluster resources. We found that the *FAIR* mode performs better regarding overall performance than the default *FIFO* mode with which later job has to wait until early job releases the resources.

Since some of the jobs in our experiments may take much longer to complete than others, we increase the Spark's max job failure value from the default 4 to 6 to give the jobs more chances to finish and hence avoid unnecessary job restart. We have also applied Spark's *KryoSerializer* tool to greatly increase the speed of serialization since a lot of feature extraction code that PIR invokes is fairly complex and lengthy.

VIII. Experiments

A. Shared-memory parallel execution

To evaluate the performance of PIR, we ran several experiments using a public Wikipedia dataset (Rasiwasia et al., 2010) consisting of 2866 Wikipedia articles (image + text) that spread over 10 categories (Table I). This dataset is about 1.53 GB in size and each article comes in a pair – every image has its corresponding text annotation. This one-to-one mapping is necessary for CCA calculation.

For evaluation of shared-memory parallel execution, we ran tests on a single machine with Intel Core I5 CPU with 2 cores and 2 hyperthreads per core so that we expect peak performance increase using a pool of 4 threads.

The experiments we ran are multi-modal query experiments. In (Clinchant, Ah-Pine, & Csurka, 2011), MIR tasks that involve the combination of two or more different modality data are summarized in three categories, early, late, and transmedia fusion. Such categorization is based mainly on when the fusion (synergy between features from various modalities) occurs. Early fusion represents the class of MIR algorithms, where different features are combined before the similarity is computed for a retrieval task while late fusion algorithms compute the similarity per modality and apply an aggregation function on the similarities to generate a final similarity. Transmedia fusion algorithm, like late fusion, obtains similarity results from different modalities / feature type first. But it differs from late fusion in that instead of the aggregation function, it applies a diffusion process to return results from a modality that is different from the query's modality. For instance, if the supplied query is an image, the query results may contain a list of text files that are "similar" to the query image.

Our experiments cover two topics – early and transmedia fusion. We skip the late fusion experiment as this is similar to transmedia fusion from execution perspective.

1) Image query with Lucene Index: The first experiment we performed is image query against index (see Listing 1 that is illustrated in Figure 8). In this experiment, we used

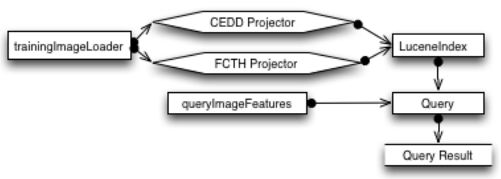

Figure 8: Image Retrieval Graph

all the 2866 images for execution. The CEDD and FTCH features were extracted from these images. The features were then fed into Lucene engine to generate index. Finally, a query image was supplied to query against the index to retrieve similar images. We showed results using the sequential vs. parallel strategy in Table II.

	image query (unit:sec)	
run	Sequential	Parallel
1	482	304
2	476	303
3	478	303
4	475	299
5	479	305

	transmedia (unit:sec)	
run	Sequential	Parallel
1	240	186
2	232	186
3	233	178
4	233	182
5	234	172

Table II: Runtime of image query and transmedia query

2) Transmedia query with CCA model: In this experiment, we performed a transmedia query multimedia information retrieval task (see Listing 2 that is illustrated in Figure 9). We used 1433 (half of the entire dataset)

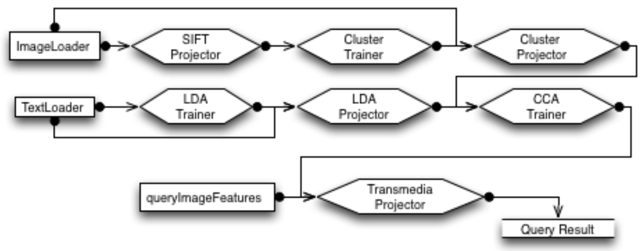

Figure 9: Transmedia Query Graph

image and text files respectively from all the 10 categories. SIFT features were extracted from the image files. Then a clustering step was applied to all the SIFT features to obtain histogram features (or bag of visual words in some literature). The LDA process was applied to the text files to obtain a LDA model. This model was then applied to all the text files to obtain a probability distribution across the 10 categories for each image. Both the histogram and distribution data from image and text were used for the CCA computation and ended up in a CCA model. Finally, a query image against the CCA model will result in a list of similar text files (with descending similarity scores) while a query text against the same CCA model will result in a list of similar image files. This is also why this is called transmedia query in the literature. The results are summarized in Table II.

The results show that our parallel strategy outperforms the sequential counterpart consistently across multiple executions. Note for both of the experiments we only parallelized portion of the pipeline when we applied the parallel strategy. Namely, only source loaders (Image and Text), the projectors (CEDD, FTCH, SIFT and LDA) were parallelized. The training and indexing processes were still sequentially executed. We should see further performance gain if we apply parallel algorithms for training and indexing.

B. Distributed execution

We evaluated the performance of PIR on Amazon EC2 with the same programs that were used in Section VIII-A.

Category	Art	Biology	Geography	History	Literature	Media	Music	Royalty	Sport	Warfare
Count	172	360	340	333	267	236	237	185	285	451

Table I: The Wikipedia article (Image + Text) data category with counts

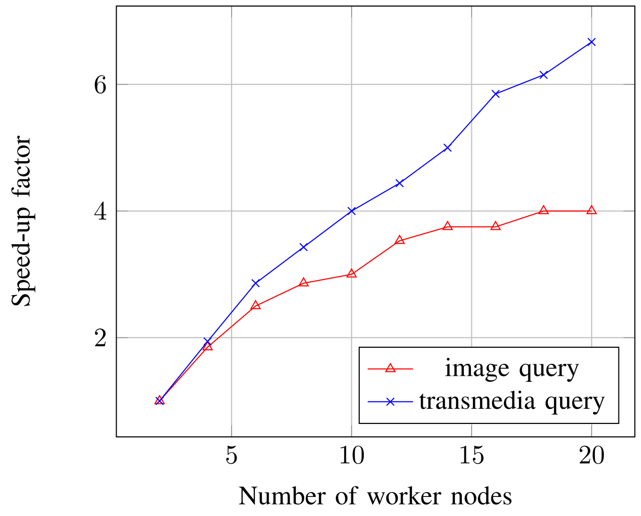

Figure 10: Scalability of image and transmedia query

Figure 10 shows the speed-up factors of runtime with 4-20 nodes over the runtime with 2 nodes. The results indicate good runtime scalability of the PIR programs running on the Amazon EC2 clusters. The total runtime decreased as the number of worker nodes increases though the performance gain is not linear. The non-linear speedup is also due to that we only parallelized portion of the PIR programs. That is, file loading (Image and Text) and feature extraction (SIFT and LDA topic distribution) are run in parallel while training (e.g. K-Means) and indexing are still sequentially. We expect to see better performance if we adopt parallel algorithms for machine learning and apply distributed indexing.

C. *Filtering and Sorting*

The last set of experiments used a larger data set and examined the performance of filtering and sorting in PIR programs. We deployed the program shown in Listing 3 on Amazon EC2 cloud. The experiments were performed using the ImageCLEF dataset IAPR TC-12 (Escalante et al., 2010) that contains 20,000 images from 40 categories. This dataset was uploaded to Amazon S3 storage bucket so that all the Amazon cluster worker nodes can access them. The pipeline process of the PIR program is illustrated in Figure 11. The pipeline uses filters to select images of the highest similarity scores in terms of the feature distance to the query image. The first filter returns 2000 images, the second filter returns 500 images, and the last filter returns 100 images. Note that the execution of a filter needs to complete to calculate the input to the next filter, which has some negative effect on the performance scaling. The runtime performance is shown

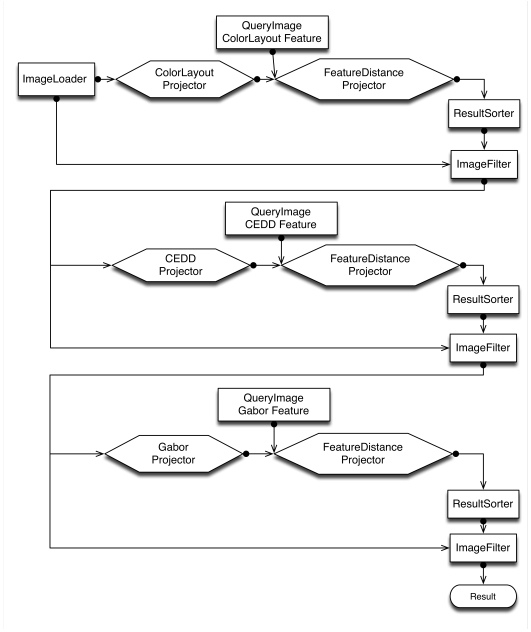

Figure 11: Series feature aggregation query graph

in Figure 12, which also demonstrates good scalability.

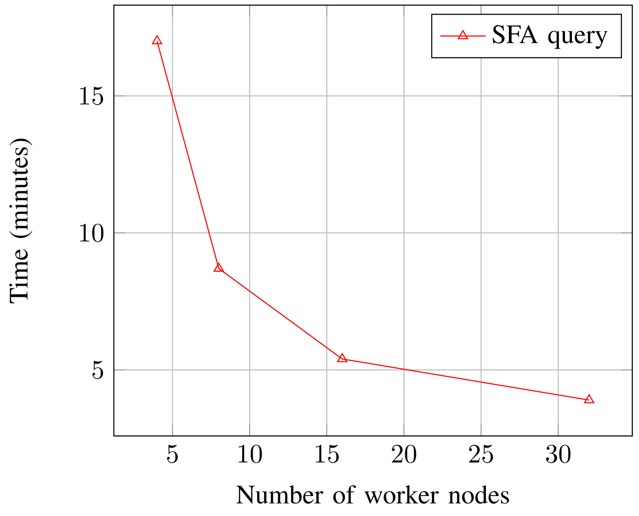

Figure 12: Runtime of series feature aggregation (SFA)

IX. Related Work

The benefits of DSL are summarized in (Spinellis, 2001), which also described some common DSL design patterns. A DSL is a special purpose language and thus needs to be designed for the relevant domain(s). Special domain knowledge and characteristics must be reflected in the DSL and in turn special operations can be adopted. PIR can be associated with several patterns mentioned in (Spinellis, 2001). Since PIR is embedded in the host language Scala, it falls under the "piggyback" pattern and naturally inherits all the merits from Scala "for free".

Below we discuss two types of related works. The first type includes DSLs that provide performance improvement by delaying the actual computation until a later stage where performance optimization can be applied. The second type includes DSLs designed to provide performance gain through domain-specific constructs for data parallel execution.

A. *Staging-based DSL*

OptiML (Sujeeth et al., 2011) is designed to provide a parallel DSL for the machine learning (ML) community to bridge the gap between ML algorithms and heterogeneous hardware. Built on top of Delite (Brown et al., 2011), OptiML performs static domain-specific optimization based on the ML algorithm features such as iterativeness and probabilistic reasoning. OptiML restricts allowed data type as one of the three basic types: Vector, Matrix, and Graph. Such restriction provides the compiler more opportunities of better code optimization. As a DSL embedded in Scala, OptiML uses a metaprogramming technique Lightweight Modular Staging to build the so-called Intermediate Representation (IR) (Rompf & Odersky, 2010) nodes. Each node contains the operation (how the data will be processed) and its dependencies (the data and control). Statements are translated to a nodes and optimization can be performed on these nodes. The "staged" IR node is similar to the vertices of our pipeline graph. However, PIR is different from OptiML in several different ways. First, PIR works for MIR domain while OptiML considers ML domain. Specifically, instead of looking for deep optimization of vector/matrix computation that ML algorithms often desire, PIR tries to help the user concentrate on building the main pipeline structure of the whole algorithm, which forms the backbone of most of the MIR tasks. Second, even though PIR is also built on top of Scala, it does not require the user to use any advanced Scala features but just connecting different types of nodes to form the pipeline. In other words, PIR users need to know very few Scala constructs. However, OptiML requires some knowledge of Scala to be able to use it effectively.

Diderot (Chiw, Kindlmann, Reppy, Samuels, & Seltzer, 2012) DSL is deigned for image analysis and visualization domain and supports a high-level model of computation that is based on continuous tensor fields. Tensor is a data structure representing the values stored in images and produced by operations on images. Tensors are designed to link the image data in the continuous and discrete space. As described in (Chiw et al., 2012), "0-order tensors, or scalars, capture real-valued samples from scans typically shown in gray-scale (e.g., CT). 1-order tensors, or vectors, describe directional quantities such as velocity and spatial derivatives of scalar fields. 2-order tensors, represented as matrices, describe linear transforms on vectors, first derivatives of vector fields, and second derivatives of scalar fields." A set of tensor operators (e.g. differentiation and convolution) are defined and special language structures, global definition (input), strand definition (computation core of the algorithm), and initialization (initial set of strands), are used to handle image analysis and visualization problems in a DSL manner. Diderot abstracted out the mathematical details from image processing domain and make that optimizable and parallelizable. In comparison, PIR emphasizes on capturing the control flow (the pipeline) in a typical multimedia information retrieval task. Thus, it is easier to locate the critical path to locate better parallelization and optimization scheme with PIR program than with Diderot.

(Christiansen, Theil Have, Torp Lassen, & Petit, 2013) presents a logic-based scripting pipeline language, BANpipe, to model compositions of time consuming analysis. BANpipe creates a pipeline to represent the relayed processing of computation in the biological sequence analysis domain. Sequence analysis often requires processing large files (input, output, and intermediate data) and these files are likely to be processed by a set of different programs. Specifically, the output of one program is the input of the next program. This naturally fits the "pipeline" paradigm. While BANpipe shares the pipeline concept with PIR, PIR is different from BANpipe in various ways. First, BANpipe works with Prolog for the biological sequence processing while PIR works with Scala in the multimedia information retrieval domain. Second, BANpipe implementation applies a list of chaining processors on files and thus essentially works like a piped batch-job processor. In comparison, PIR implementation follows two steps, translating code into a pipeline graph and implements the graph with different strategies. With such high level pipeline graph abstraction, different optimization, scheduling, and analyzing can be performed on the graph level independent of the detailed operations and/or computation within each node. In addition, PIR provides a much clearer data and control flow and thus is very domain user friendly (allow them quickly modifying program without worrying about the underneath implementation).

B. *Data-parallel DSL*

FlumeJava (Chambers et al., 2010) provides a Java library to support data-parallel pipelines. It defers evaluations of operations and constructs an execution plan dataflow graph. This dataflow graph serves similar purposes (e.g. optimization) as the PIR pipeline graph. However, PIR pipeline graph focuses on different MIR tasks while FlumeJava is for general computation.

DryadLINQ (Yu et al., 2008) translates .NET applications into a distributed execution plan, which contains DryadLINQ subexpressions. Each subexpression is executed in a separate Dryad vertex. A job manager creates a job graph using the execution plan and spawns vertices as resources become available on the cluster. Each vertex executes its program and writes data to the output table. This design gives the users choices to continue writing traditional high-level imperative program. However, such strength comes with a cost. The underlying distributed environment is supported by a compiler and code generator, which increases the overall execution time.

There are several other popular DSL languages such as Google's Sawzall (Pike, Dorward, Griesemer, & Quinlan, 2005), Yahoo's Pig (Olston, Reed, Srivastava, Kumar, & Tomkins, 2008), and Facebook's Hive (Thusoo et al., 2009)) that allow the developers to write declarative programs on top of distributed computation platforms. For example, Sawzall, Pig, and Hive code are all compiled to general MapReduce applications. Compared to these DSLs, PIR has a very small footprint as an embedded DSL. PIR's "compile" process is just part of the runtime execution. Also, PIR can be easily integrated with different computing environments by switching to different execution strategies. For example, the Spark integration code has less than 100 lines of code in total. As a MIR DSL, we have specialized constructs that serve MIR tasks well and can be optimized incrementally in future versions.

X. Conclusion

In this paper, we presented a domain specific language PIR for implementing MIR applications. PIR provides a high level structure for constructing MIR workflow and hides the details of resource management, optimization, and parallelization. Programs written in PIR are easy to read and reusable. It is also extensible since the DSL is based on a Java compatible host language Scala that allows us to incorporate algorithms implemented in Java-based MIR libraries. As future work, we plan to investigate the applicability of our DSL to wider range of MIR applications.

References

Amazon. (2013). *Amazon elastic compute cloud (amazon ec2).* Retrieved from `http://aws.amazon.com/ec2/`

Atrey, P. K., Hossain, M. A., El Saddik, A., & Kankanhalli, M. S. (2010). Multimodal fusion for multimedia analysis: a survey. *Multimedia Systems*, *16*(6), 345-379.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: Speeded up robust features. In *Computer vision–eccv 2006* (pp. 404–417). Springer.

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *the Journal of machine Learning research*, *3*, 993–1022.

Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M., & Olukotun, K. (2011). A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 international conference on parallel architectures and compilation techniques* (pp. 89–100).

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., & Weizenbaum, N. (2010). FlumeJava: Easy, efficient data-parallel pipelines. *SIGPLAN Not.*, *45*(6), 363–375.

Chatzichristofis, S. A., & Boutalis, Y. S. (2008a). Cedd: Color and edge directivity descriptor: A compact descriptor for image indexing and retrieval. In *Computer vision systems* (pp. 312–322). Springer.

Chatzichristofis, S. A., & Boutalis, Y. S. (2008b). Fcth: Fuzzy color and texture histogram-a low level feature for accurate image retrieval. In *Image analysis for multimedia interactive services, 2008. wiamis'08. ninth international workshop on* (pp. 191–196).

Chiw, C., Kindlmann, G., Reppy, J., Samuels, L., & Seltzer, N. (2012). Diderot: a parallel dsl for image analysis and visualization. In *Proceedings of the 33rd acm conference on programming language design and implementation* (pp. 111–120).

Christiansen, H., Theil Have, C., Torp Lassen, O., & Petit, M. (2013). A declarative pipeline language for complex data analysis. In *Logic-based program synthesis and transformation* (p. 17-34).

Clinchant, S., Ah-Pine, J., & Csurka, G. (n.d.). Semantic combination of textual and visual information in multimedia retrieval. In *Proceedings of the 1st acm international conference on multimedia retrieval* (p. 44). ACM.

Clinchant, S., Ah-Pine, J., & Csurka, G. (2011). Semantic combination of textual and visual information in multimedia retrieval. In *International conference on multimedia retrieval.*

CVML. (2007). *The gift (the gnu image-finding tool), made by the vision group at the cui (computer science center) of the university of geneva.* (https://www.gnu.org/software/gift/)

Datta, R., Joshi, D., Li, J., & Wang, J. Z. (2008). Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys*, *40*(2).

Deselaers, T., Keysers, D., Hegerath, A., Weyand, T., Velroyen, H., Forster, J., ... Gass, T. (2010). *Fire, the flexible image retrieval engine.* (https://code.google.com/p/fire-cbir/)

Dumais, S. T. (2004). Latent semantic analysis. *Annual review of information science and technology*, *38*(1), 188–230.

Escalante, H. J., Hernández, C. A., Gonzalez, J. A., López-López, A., Montes, M., Morales, E. F., ... Grubinger, M. (2010). The segmented and annotated iapr tc-12 benchmark. *Computer Vision and Image Understanding*, *114*(4), 419–428.

Hare, J. S., Samangooei, S., & Dupplaw, D. P. (n.d.). Openimaj and imageterrier: Java libraries and tools for scalable multimedia analysis and indexing of images. In *Proceedings of the 19th acm international conference on multimedia* (p. 691-694).

Hare, J. S., Samangooei, S., Dupplaw, D. P., & Lewis, P. H. (n.d.). Imageterrier: an extensible platform for scalable high-performance image retrieval. In *Proceedings of the 2nd acm international conference on multimedia retrieval* (p. 40).

Hartigan, J. A., & Wong, M. A. (1979). Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, 100–108.

Hu, W., Xie, N., Li, L., Zeng, X., & Maybank, S. (2011). A survey on visual content-based video indexing and retrieval. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, *41*(6), 797-819.

Lew, M. S. (2012). Multimedia information retrieval in the twenty-first century. *International Journal of Multimedia Information Retrieval*, *1*(1), 1-2.

Lew, M. S., Sebe, N., Djeraba, C., & Jain, R. (2006). Content-based multimedia information retrieval: State of the art and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications*, *2*(1), 1-19.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision (IJCV)*, *60*(2), 91-110.

Lux, M. (2009). *Caliph & emir: Mpeg-7 based java prototypes for digital photo and image annotation and retrieval.* (http://www.semanticmetadata.net/features/)

Manjunath, B. S., Ohm, J.-R., Vasudevan, V. V., & Yamada, A. (2001). Color and texture descriptors. *Circuits and Systems for Video Technology, IEEE Transactions on*, *11*(6), 703–715.

Matas, J., Chum, O., Urban, M., & Pajdla, T. (2004). Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, *22*(10), 761–767.

McCallum, & Kachites, A. (2002). *Mallet: A machine learning for language toolkit.* Retrieved from `http://mallet.cs.umass.edu/`

Mohamed, H., & Marchand-Maillet, S. (2012). Distributed media indexing based on mpi and map-reduce. In *International workshop on content-based multimedia indexing.*

Moise, D., Shestakov, D., Gudmundsson, G., & Amsaleg, L. (n.d.). Indexing and searching 100m images with map-reduce. In *Proceedings of the 3rd acm conference on international conference on multimedia retrieval* (p. 17-24). ACM.

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008). Pig latin: A not-so-foreign language for data processing. In *Proceedings of the international conference on management of data.*

Pike, R., Dorward, S., Griesemer, R., & Quinlan, S. (2005, October). Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, *13*(4), 277–298.

Rasiwasia, N., Costa Pereira, J., Coviello, E., Doyle, G., Lanckriet, G., Levy, R., & Vasconcelos, N. (2010). A New Approach to Cross-Modal Multimedia Retrieval. In *Acm international conference on multimedia.*

Rompf, T., & Odersky, M. (2010, October). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *SIGPLAN Not.*, *46*(2), 127–136.

Savvas, L. M., & Chatzichristofis, A. (2008). LIRE: Lucene Image Retrieval ?? An Extensible Java CBIR Library. In *16th acm international conference on multimedia.*

Scherp, A., & Mezaris, V. (2013). Survey on modeling and indexing events in multimedia. *Multimedia Tools and Applications*, 1-17.

Shestakov, D., Moise, D., Gudmundsson, G. T., & Amsaleg, L. (n.d.). Scalable high-dimensional indexing with hadoop. In *International workshop on content-based multimedia indexing.*

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *J. Syst. Softw.*, *56*(1), 91–99.

Sujeeth, A. K., Lee, H., Brown, K. J., Chafi, H., Wu, M., Atreya, A. R., … Odersky, M. (2011). Optiml: an implicitly parallel domainspecific language for machine learning. In *in proceedings of the 28th international conference on machine learning.*

Thompson, B. (2005). Canonical correlation analysis. *Encyclopedia of statistics in behavioral science*.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., … Murthy, R. (2009, August). Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, *2*(2).

Tuytelaars, T., & Mikolajczyk, K. (2008). Local invariant feature detectors: a survey. *Foundations and Trends? in Computer Graphics and Vision*, *3*(3), 177-280.

VanDrunen, T., & Palsberg, J. (2004). Visitor-oriented programming. In *Proceedings of fool-11, the 11th acm sigplan international workshop on foundations of object-oriented languages.*

Wang, M., & Hua, X.-S. (2011). Active learning in multimedia annotation and retrieval: A survey. *ACM Transactions on Intelligent Systems and Technology*, *2*(2), 10.

Yoshitaka, A., & Ichikawa, T. (1999). A survey on content-based retrieval for multimedia databases. *IEEE Transactions on Knowledge and Data Engineering*, *11*(1), 81-93.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., & Currey, J. (2008). DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th usenix conference on operating systems design and implementation.*

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *Hotcloud 2010.*

Zhang, J., & Ye, L. (2010, July). Series feature aggregation for content-based image retrieval. *Comput. Electr. Eng.*, *36*(4), 691–701.