1. Introduction

- The Pig system and the Pig Latin programming language were first proposed in 2008 in a top-tier database research conference:
  - Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: a not-so-foreign language for data processing. ACM SIGMOD Conference 2008: 1099-1110
- Most examples in this module are courtesy Chris Olston and Utkarsh Srivastava, both co-authors on the SIGMOD paper.
- While Pig started out as a research project at Yahoo! Research, it is now an Apache open source project with an active community of contributors.

1.1 Overview

- Pig's design goal was to find a sweet spot between "plain" MapReduce, where all functionality is expressed in Java or C++ in a low-level procedural style, and the declarative style of SQL.
- The programmer creates a Pig Latin program using high-level pre-defined operators. Additional functionality can be added through user-defined functions (UDF).
- A Pig Latin program is compiled to a MapReduce program, possibly consisting of multiple jobs, to run on Hadoop.

1.2 Why Not Use SQL or Plain MapReduce?

- SQL, which is the standard query language for relational databases, has been very successful for big data analysis for many decades. So, why invent the new Pig Latin language?
  - Many programmers and data analysts find it difficult to create and debug SQL queries. In SQL, a query specifies WHAT the user is looking for through constraints over database tables. This way of thinking about a program is less familiar to many people than the procedural way of specifying step-by-step HOW to derive the result. A small mistake in an SQL constraint could completely change the result; and there is no good way to perform step-by-step debugging as in procedural languages like Java and C++.
  - One of the reasons for the success of relational databases and SQL is that they enable the use of automatic optimizers for finding a good query plan implementation for a given query. Unfortunately, automatic optimizers can make mistakes, sometimes choosing an inefficient implementation. For this reason, when dealing with big data, some programmers do not fully trust an automatic optimizer and might prefer to hard-code what they believe is the best query plan.

1.2 Why Not Use SQL or Plain MapReduce?

- Plain MapReduce lacks the convenience of readily available and reusable data manipulation operators like selection, projection, join, and sort. It is much faster and less error-prone to compose complex data analysis programs from such operators. And since program semantics are hidden in complex Java code, plain MapReduce programs are more difficult to optimize and maintain.
- Pig Latin attempts to combine the best of both worlds by taking the following approach:
  - Like relational databases, Pig Latin relies on high-level operators to process data. In fact, many Pig Latin operators are versions of well-understood relational operators.
  - Like plain MapReduce, a Pig Latin program specifies a sequence of data manipulation steps.
  - The Pig system comes with an automatic optimizer. To avoid potentially costly mistakes, this optimizer only applies automatic transformations that are considered “safe.” For example, it is generally a good idea to filter out irrelevant records or fields of records as early as possible.

1.3 Example Data Analysis Task

Find the top-10 most visited pages in each category

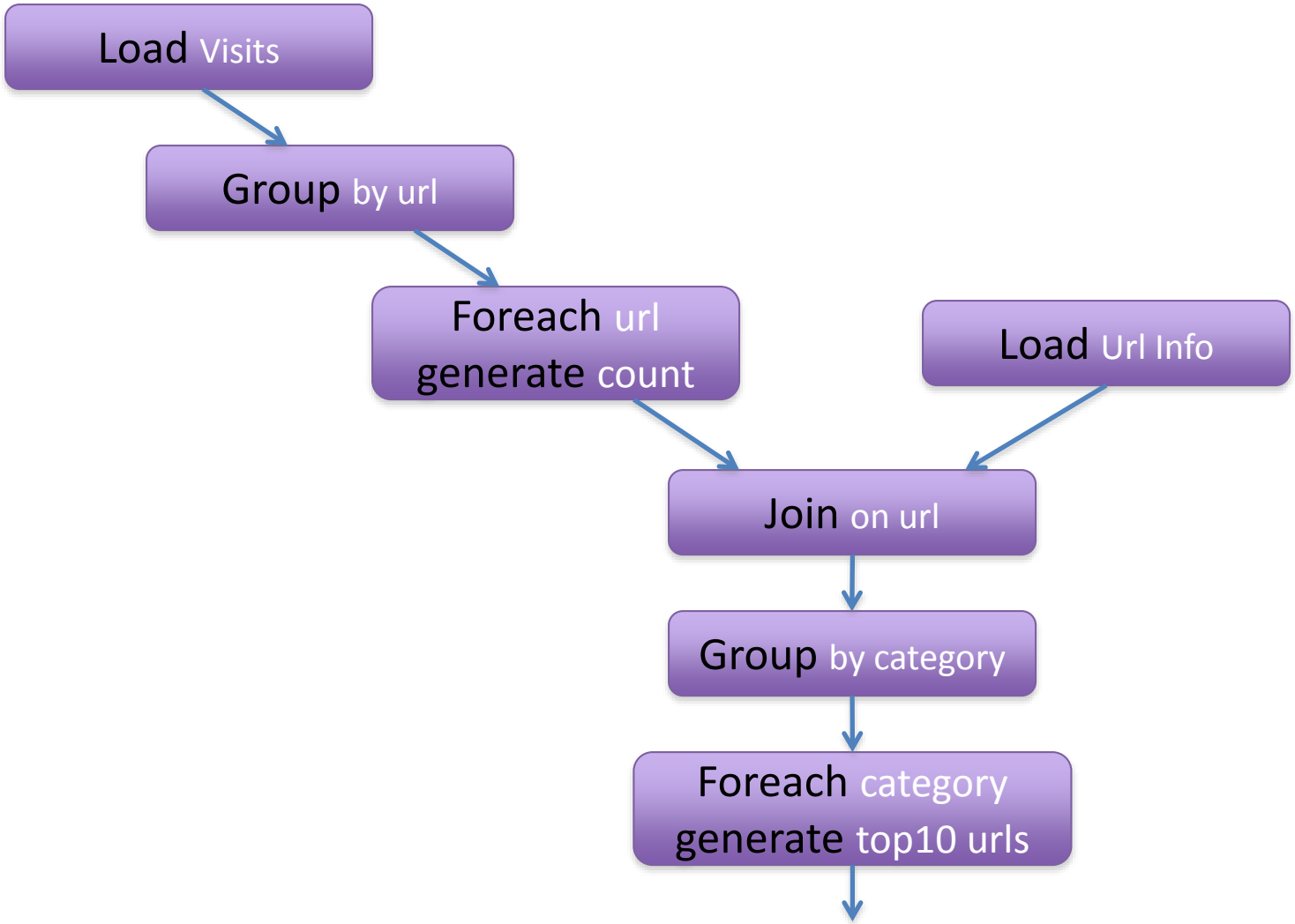
Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

URL Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9

1.3.1 Schematic Data Flow



1.3.2 Implementation In Pig Latin

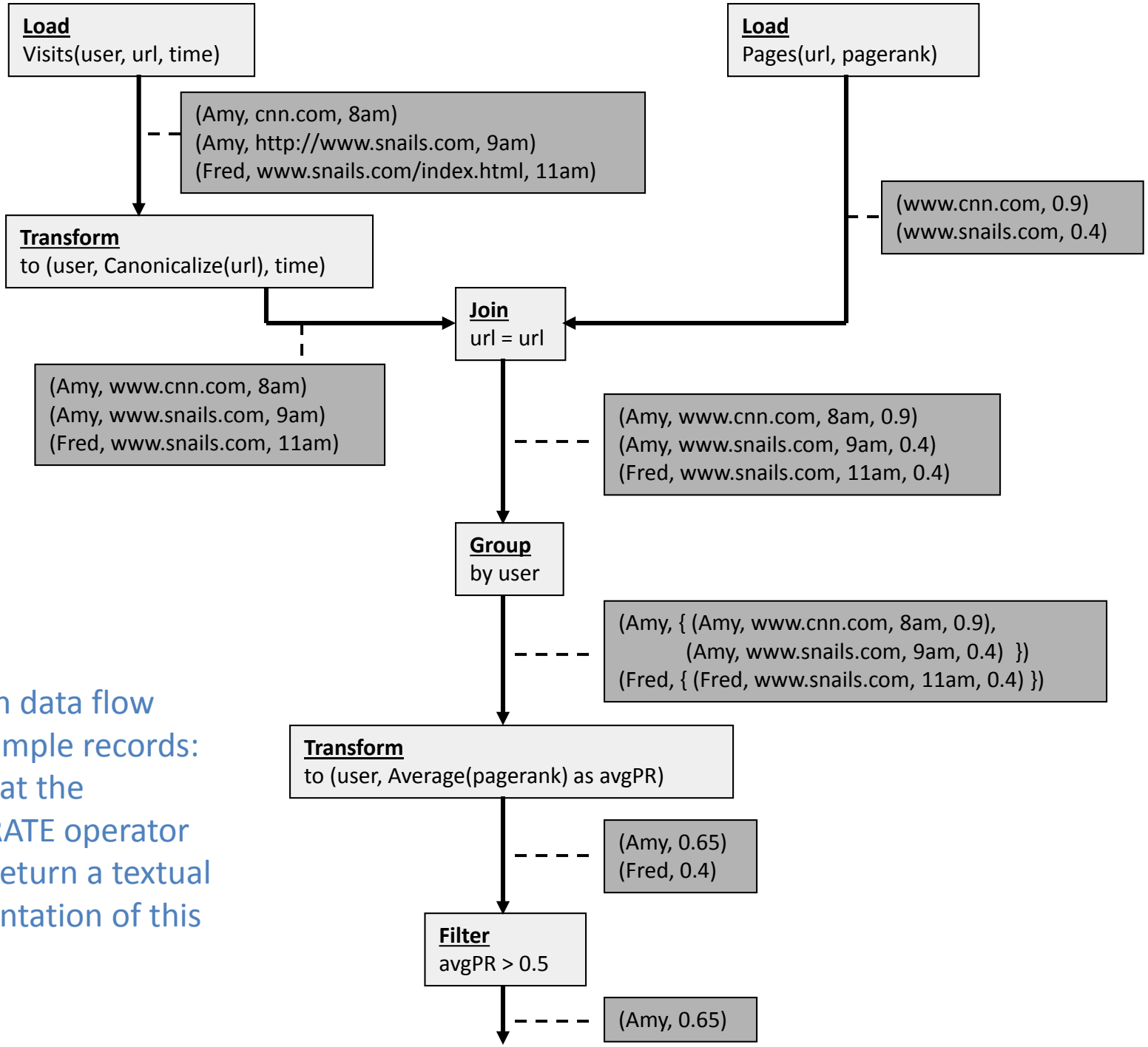
```
visits       = load '/data/visits' as (user, url, time);
gVisits      = group visits by url;
visitCounts  = foreach gVisits generate url, count(visits);

urlInfo      = load '/data/urlInfo' as (url, category, pRank);
vCountsCat   = join visitCounts by url, urlInfo by url;

gCategories = group vCountsCat by category;
topUrls = foreach gCategories generate top(vCountsCat,10);

store topUrls into '/data/topUrls';
```

1.4 Running Pig

- Pig can run in *local mode* or in *MapReduce mode*. The former is useful for testing and debugging, using a single machine. The latter executes the program on a Hadoop cluster.
- One can also chose between *interactive mode* and *batch mode*. For the former, Pig's Grunt shell is used to enter Pig Latin statements one-by-one. The latter executes a Pig Latin program using the *pig* command.
- Pig Latin execution is *lazy* in the sense that no statement will executed until it has to. In particular, if there is no DUMP or STORE operator forcing the program to output results, then no data loading or processing will be executed. This makes sense, because there is no need to waste resources unless the user is interested in the result.
- There is no need to import data into a database or other type of repository. Pig Latin reads directly from files. Schemas are optional and can be assigned dynamically. For example, "LOAD '/data/visits' as (user, url, time);" assigns schema (user, url, time) to the tuples loaded from the Visits file.
- In a Pig Latin program, user-defined functions can be called in every construct such as LOAD, STORE, GROUP, FILTER, and FOREACH. For example, in "FOREACH gCategories GENERATE top(visitCounts,10);", the top() function is user-defined, typically in a language like Java or Python.

1.4 Running Pig

- Pig has several operators to simplify the debugging process:
  - DUMP displays (intermediate) results to the terminal screen.
    - "DUMP gVisits;" would reveal the content of the grouped Visits log.
  - DESCRIBE shows the schema of a data set.
  - EXPLAIN displays the logical, physical, and MapReduce plan used for computing the specified (intermediate) result.
    - The logical plan shows the pipeline of logical, i.e., backend-independent, operators. If backend-independent optimizations were applied, e.g., applying filters early, they would be reflected here.
    - The physical plan shows the implementation of the logical operators by backend-specific physical operators. Backend-specific optimizations would be reflected here.
    - The MapReduce plan shows how the physical operators are grouped into MapReduce jobs.
  - ILLUSTRATE displays the step-by-step execution for computing the specified (intermediate) result. The steps are accompanied by example tuples that make it easier to spot errors. To be able to show examples, the LOAD statements must define a schema. [Link: Illustration of a data flow program with examples.]



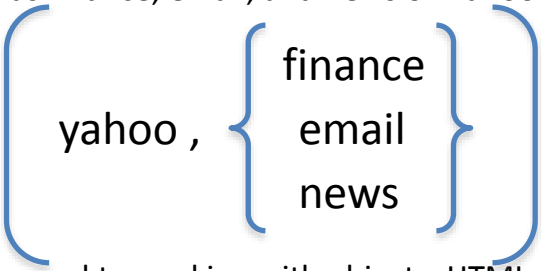
Pig Latin data flow and example records: Note that the ILLUSTRATE operator would return a textual representation of this graph.

2. The Pig Latin Language

- Let's look at the Pig Latin language in more detail. Note that as Pig is continuously being updated and improved, details keep changing and features are being added. For up-to-date information, refer to the documentation of the Pig Latin version you are working with. In this module we provide only an overview of the most important features and functionality, which tends to remain stable.

2.1 Pig Latin Data Model

- Data comes in all kinds of formats:
  - Relational databases represent data in “flat” tables. In particular, the different attributes of a database tuple are simple scalar types such as integer, floating point number, or strings. Sets or lists are not allowed as attribute values.
  - HTML and XML documents have a nested structure, i.e., an element can contain other elements.
  - Objects in programing languages like Java can contain simple scalar types or other objects.
  - Text documents often have no explicit structure at all, except based on formatting (sections, paragraphs, sentences).
- To support all these types of structured and semi-structured data, Pig Latin relies on a fully-nestable data model with atomic values (e.g., strings and numbers), tuples (a sequence of fields), bags (a collection of tuples that allows for duplicates), and maps (a collection of key-value pairs). For example, one can represent the offerings in categories such as finance, email, and news on Yahoo’s site as follows:



- To programmers used to working with objects, HTML, or XML, this model is more familiar and natural than data representation through flat tuples in a relational database.
  - Note that it would be conceptually straightforward to generate flat tuples from a nested tuple. The above example would be flattened into three distinct tuples: (yahoo, finance), (yahoo, email), and (yahoo, news). However, depending on the nesting level and number of values in each level, flattening could be expensive as all possible combinations need to be generated. Pig Latin’s FLATTEN operator supports this functionality.
- The downside of the convenience of the nested data model is a potential performance cost. Databases have been successful because once the data is represented in comparably simple flat tables, it is much easier to reason about query performance and to find efficient implementations. In contrast, nested data such as Java objects are often expensive to process, e.g., when computation requires pointer traversals through complex structures.

2.2 Pig Latin Operators

- A Pig Latin program typically has the following structure:
  - Load data from input files.
  - Process the data and possibly store intermediate results.
  - Store final results.

2.2.1 Loading Data: LOAD

- LOAD reads data from a file or directory and optionally assigns a schema to each tuple. A schema consists of field names and (optionally) their types.
- To deal with specific file formats, a custom deserializer can be used.
- For example, consider a log of Web search queries. When a user submits a search query, e.g., “Northeastern University”, a log record is added containing user ID, query string, and the time the query was submitted. To process this log file, it first has to be loaded:
  - queries = LOAD ‘query_log.txt’ USING myLoad() AS (userID, queryString, timestamp);

2.2.2 Processing Data: FOREACH

- The FOREACH operator applies some processing to each tuple of a bag. Since each tuple is processed individually and independent of the other ones, computation can be parallelized easily. [Challenge question link]
- GENERATE specifies how to convert the input tuple to an output tuple.
- In the log analysis example, assume each original user query string is first expanded, e.g., by adding the manufacturer's name to a product. Function expandQuery() is user-defined and could return a bag, instead of just a single string:
  - expanded_queries = FOREACH queries GENERATE userId, expandQuery(queryString);

2.2.2 Processing Data: FILTER

- The FILTER operator discards unwanted tuples, which do not pass the filter condition. The filter condition can involve standard comparison operators and user-defined functions.
- To eliminate log entries for queries from non-human users ("bots"), the following filter could be used:
  - Simple approach with standard comparison operator neq (for inequality test):
    - real_queries = FILTER queries BY userId neq `bot';
  - More sophisticated approach with a user-defined function that might rely on a data mining model to identify bot queries:
    - real_queries = FILTER queries BY NOT isBot(userId);

2.2.2 Processing Data: COGROUP

- The COGROUP operator groups multiple input bags by some of their fields. It is best understood through an example. Consider data about results of Web searches (consisting of the query string and the ranks of the top URLs) and the ad revenue for these search queries (specifying the amount depending on where the ad is placed). The goal is to organize the data in such a way that for each search query, there is the list of top-ranked URLs and the list of all possible ad revenue amounts. COGROUP computes the desired output:

results

queryString	url	rank
Lakers	nba.com	1
Lakers	espn.com	2
Kings	nhl.com	1
Kings	nba.com	2

revenue

queryString	adSlot	amount
Lakers	top	50
Lakers	side	20
Kings	top	30
Kings	side	10

COGROUP results BY queryString, revenue BY queryString

Output:

(Lakers, { (Lakers, nba.com, 1), (Lakers, espn.com, 2) }, { (Lakers, top, 50), (Lakers, side, 20) })

(Kings, { (Kings, nhl.com, 1), (Kings, nba.com, 2) }, { (Kings, top, 30), (Kings, side, 10) })

The first field of the result schema is called “group”, identifying the group the records belong to. In the example, there are two groups: one for Lakers and the other for Kings.

2.2.2 Processing Data: GROUP

- GROUP was introduced as a special case of COGROUP, to group a single bag by the selected field(s). It is similar to GROUP BY in SQL, but does not need to apply an aggregate function to the tuples in each group. Hence it works more like a data partitioning operator.
  - Note that in Pig Latin 0.12 operators GROUP and COGROUP are identical and could be used interchangeably.
- Example for grouping the ad revenue information by search query string:
  - grouped_revenue = GROUP revenue BY queryString;

2.2.2 Processing Data: JOIN

- The JOIN operator computes an equi-join between two or more bags on the specified fields. Pig Latin supports inner and (left, right, and full) outer join. The programmer can choose from a variety of join implementations, including Replicated join, and can also set the number of Reduce tasks to be used.
- Example for joining the search results with the revenue relation on the queryString field:
  - join_result = JOIN results BY queryString, revenue BY queryString;
- Note that JOIN is just a syntactic shorthand for COGROUP followed by flattening. The above join could be written as:
  - temp_var = COGROUP results BY queryString, revenue BY queryString;
  - join_result = FOREACH temp_var GENERATE FLATTEN(results), FLATTEN(revenue);

2.2.3 Other Commonly Used Pig Latin Operators

- UNION: computes the union of two or more bags.
- CROSS: computes the cross product of two or more bags.
- ORDER: sorts a bag by the specified field(s).
- DISTINCT: eliminates duplicate tuples in a bag.
- STORE: saves a bag to a file.
- Note that nested bags within tuples can be processed by also nesting the operators within a FOREACH operator.

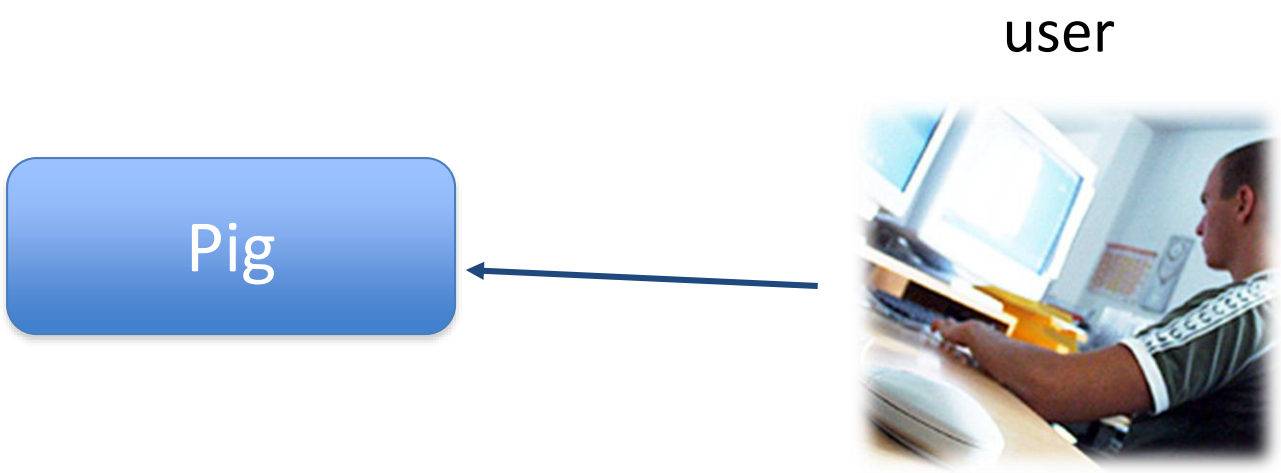
2.2.4 Fun Fact: MapReduce in Pig Latin

- As a fun exercise, let's see how we can implement any MapReduce program in Pig Latin. Let map() and reduce() be the given Map and Reduce function, respectively. Then the following Pig Latin program implements the desired MapReduce program:
  - map_result = FOREACH input GENERATE FLATTEN(map(*));
  - key_groups = GROUP map_result BY $0;
  - output = FOREACH key_groups GENERATE reduce(*);
- Here map() and reduce() are called as user-defined functions (UDF) by the Pig Latin program. The "*" indicates that the entire input record is passed to the UDF.
- $0 refers to the first field of a tuple, i.e., the intermediate key in the example. (Recall that Map emits (key, value) pairs.)

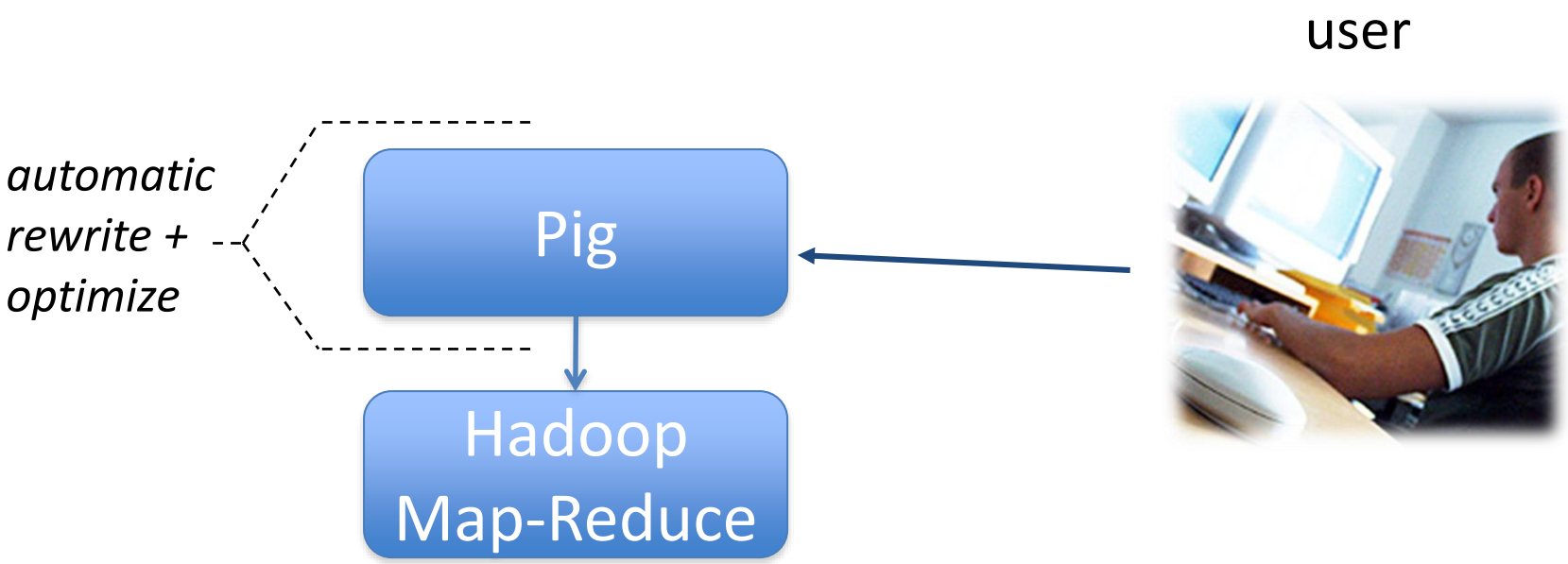
3. Pig System and Implementation

- Let's see how a programmer interacts with Pig and what Pig does internally with a given program.

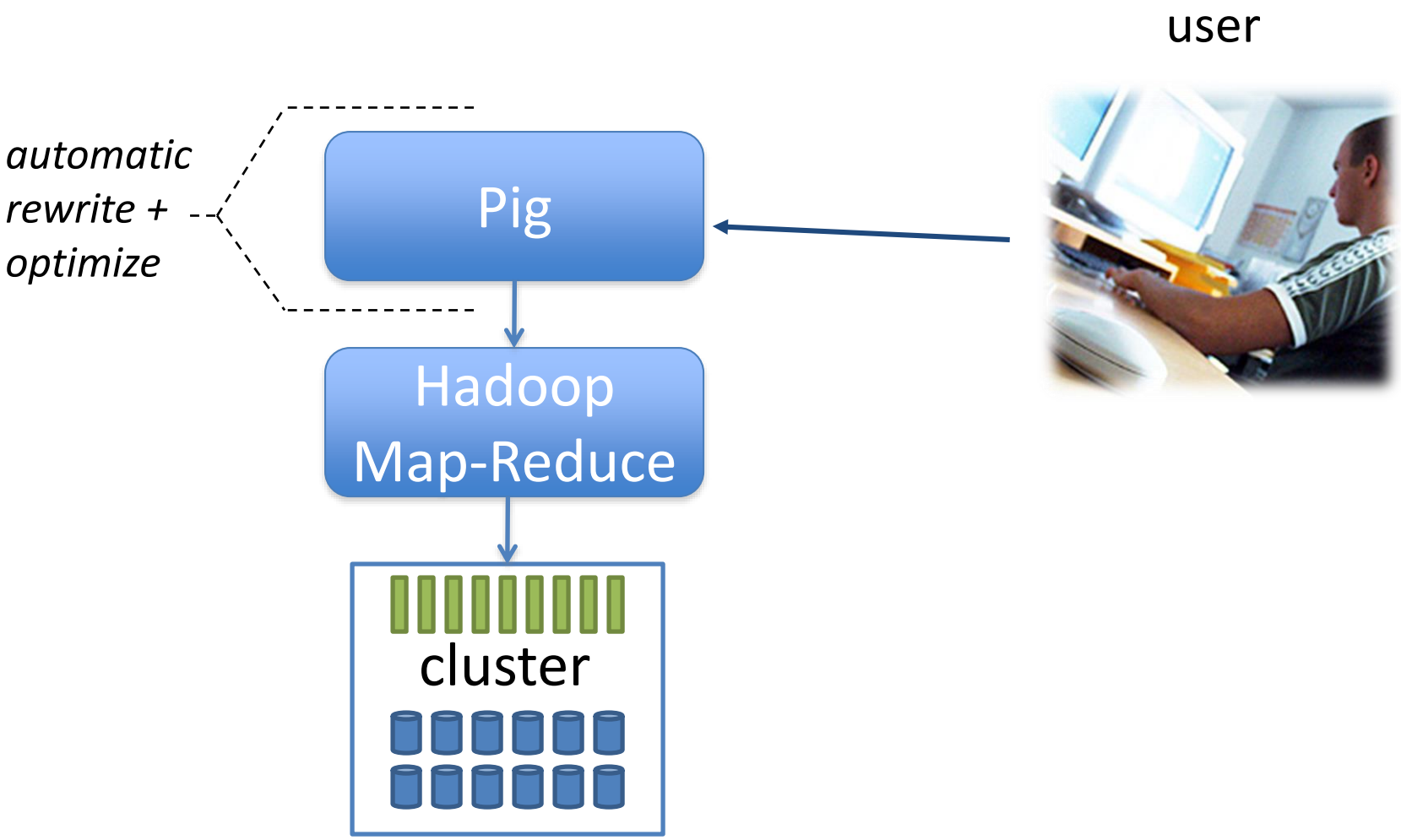
3.1 Pig Implementation



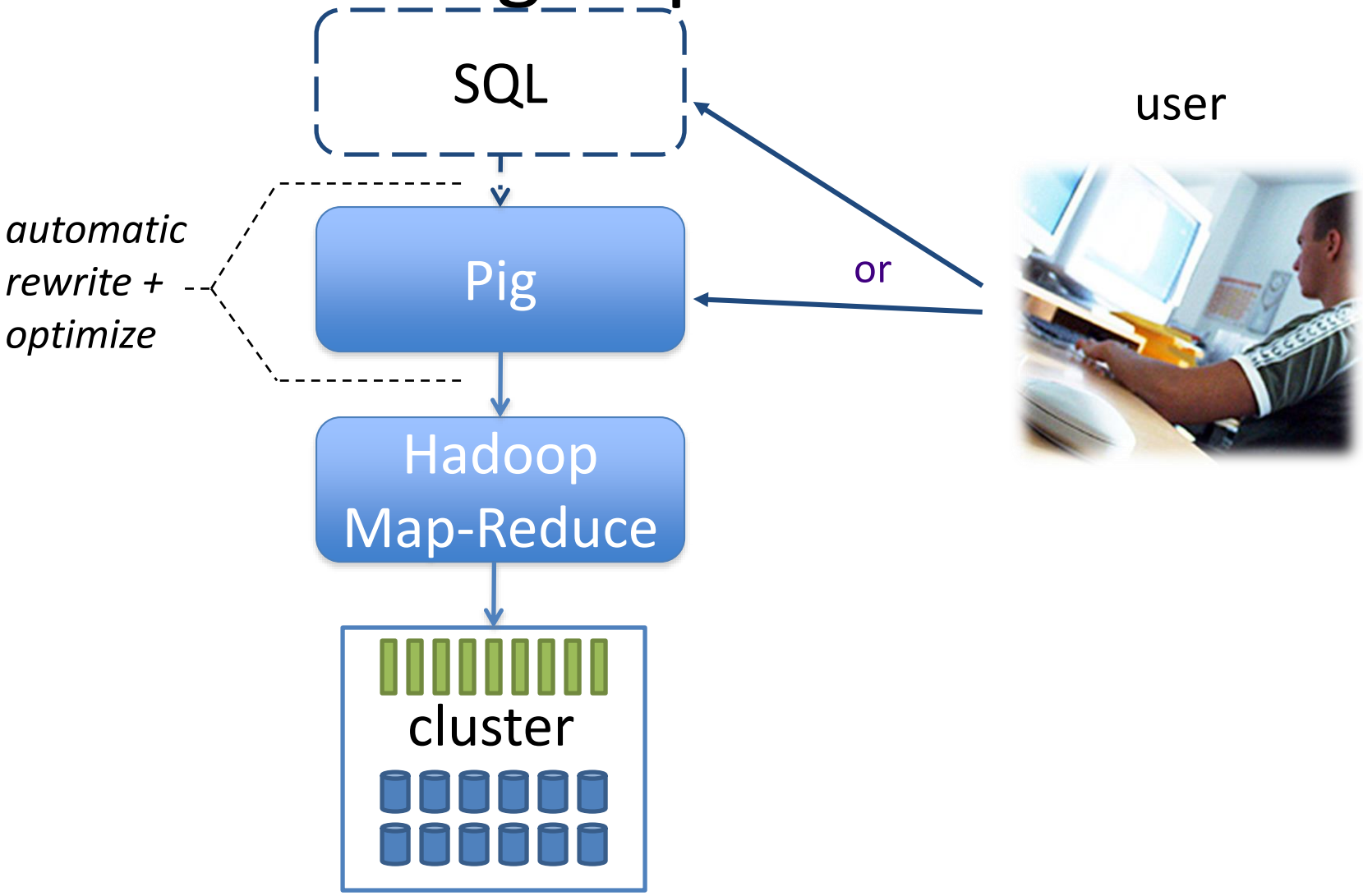
3.1 Pig Implementation



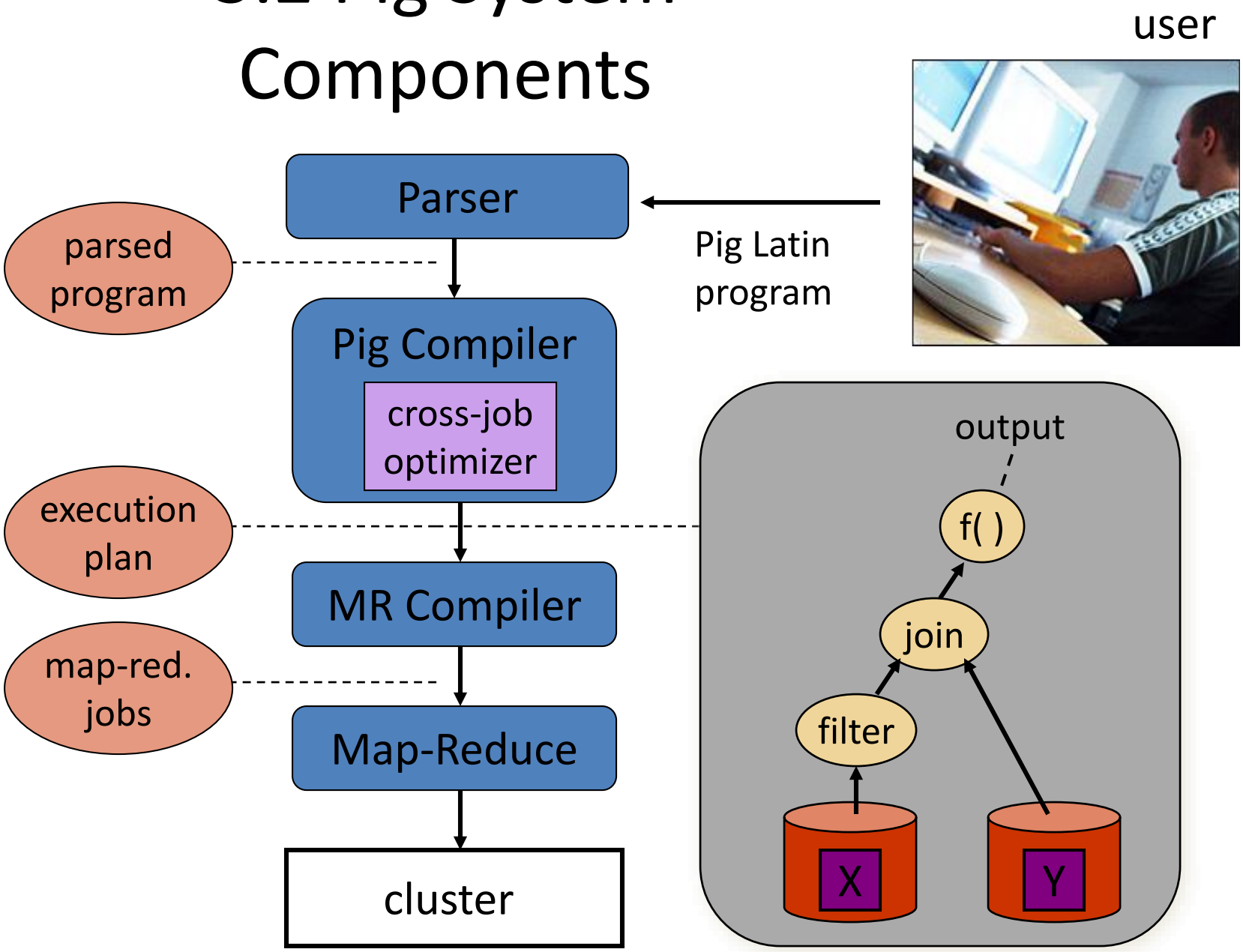
3.1 Pig Implementation



3.1 Pig Implementation



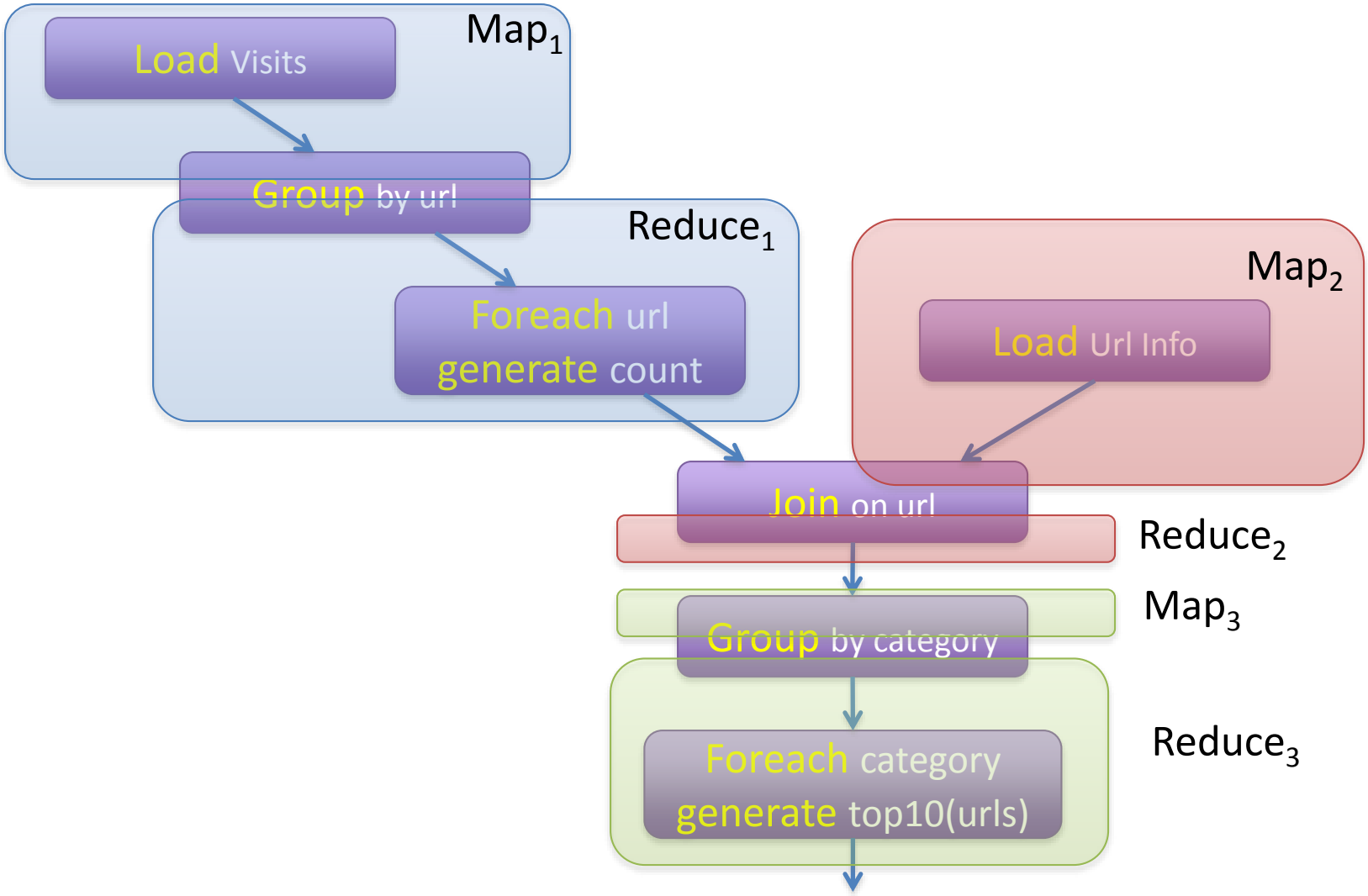
3.2 Pig System Components



3.3 Pig Optimizer

- The Pig optimizer can automatically rewrite a given program based on “safe” rules that virtually always improve performance in practice. These optimization rules are applied by default and most can be disabled.
- Example optimization rules:
  - Simplification of conditions, e.g., replaces condition “((x > 0) OR (x > 1 AND y > 2)” by the equivalent simpler condition “x > 0”
  - Pushing filters “upstream”, i.e., to operators performed earlier, to avoid passing along irrelevant tuples and fields. This is particularly important for joins. For instance, first joining Visits and URL Info on URL and then only selecting tuples for the “News” category would unnecessarily join tuples for other categories such as “Photos” and “Sports”, just to discard them soon after. Instead, it would be much cheaper to first eliminate the tuples for irrelevant categories in URL Info, and then perform the join afterward on the smaller data.
  - The Pig optimizer can also split some conditions so that they can be pushed further. Consider condition “User = ‘Amy’ AND Category = ‘News’” on the output of the join of Visits and URL Info. This condition cannot directly be applied to Visits (because it does not have a Category field) or URL Info (because it does not have a User field). However, one can equivalently apply partial filters “User = ‘Amy’” to Visits and “Category = ‘News’” to URL Info before the join, achieving the exact same result as by applying the full filter after the join. (Note that this would not have worked if the condition was a disjunction, e.g., “User = ‘Amy’ OR Category = ‘News’”.)
  - Applying LIMIT conditions early. “LIMIT k” specifies that the user does not require the entire output of an operator, but only k tuples. Pushing this condition up can also reduce intermediate result size, reducing processing cost.
- If optimizations were applied, they are reflected in the output of the EXPLAIN operator.

3.4 Compilation into MapReduce



3.4 Compilation into MapReduce

- How many MapReduce jobs will be generated for a Pig Latin program?
- That depends on the number of operators that require re-shuffling or aggregation of data. By default, each group or join operation forms a MapReduce boundary. Operators performing per-tuple computation are pipelined into a “neighboring” Map or Reduce phase.
- Recall the data flow for the top-10 most visited URLs per category. The first MapReduce job is defined around the [Group by URL] operator. LOAD is pipelined into the Map phase, while the per-group counting is pipelined into the Reduce phase. The join on URL with the URL Info data requires another MapReduce job. The following grouping by category requires re-shuffling of the joined tuples, requiring a third MapReduce job. The per-category operation can then be included in the Reduce phase of this job.

4. Summary

- Pig enables programmers to write concise data processing code that is automatically compiled to plain MapReduce, executable on Hadoop. Development time and code complexity are generally reduced significantly compared to plain MapReduce.
- Pig Latin's operators support a great variety of common analysis tasks. However, they provide less fine-grained control over data processing compared to using Java.
- In practice, Hadoop code generated from a Pig Latin program tends to have performance similar to a well-written plain Hadoop program. In some cases an experienced programmer might squeeze out more performance through careful coding in plain MapReduce, in other cases Pig's optimizer might win by discovering optimizations overlooked by the programmer.
- In summary, Pig adds a convenient layer on top of MapReduce for rapid development of efficient parallel programs.
- How does Pig compare to relational databases, another approach with a concise language for big data processing tasks?
  - [Insert the table from the next slide here.]

Pig versus Relational Database Systems

	Relational Database System	Pig
workload	Good at performing bulk, as well as random reads and writes. Indexes support fast random access. Transactions enable updates of database tables in the presence of multiple readers and writers.	Designed for bulk reads and writes only. There are no indexes for fast random access. There are no updates of existing files: output is written to new files.
data representation	The data is "flat" and the database schema (attributes and their types) must be pre-declared. Data has to be imported before use; the database controls and enforces the data format.	"Pigs eat anything." Schemas are optional and can be assigned on-the-fly when reading data from file. Data can be nested.
programming style	Specify WHAT you want: An SQL query is a system of constraints. The database optimizer then decides automatically how to efficiently compute the desired result.	Specify HOW to compute it: A Pig Latin program defines a sequence of data-processing steps. However, an optimizer might apply "safe" transformations to improve performance.
customizable processing	Custom functions can be used, but are second-class to logic expressions.	It is easy to incorporate custom functions.