Big Data Programming Models

Dongyao Wu, Sherif Sakr and Liming Zhu

Abstract Big Data programming models represent the style of programming and present the interfaces paradigm for developers to write big data applications and programs. Programming models normally the core feature of big data frameworks as they implicitly affects the execution model of big data processing engines and also drives the way for users to express and construct the big data applications and programs. In this chapter, we comprehensively investigate different programming models for big data frameworks with comparison and concrete code examples.

A programming model is the fundamental style and interfaces for developers to write computing programs and applications. In big data programming, users focus on writing data-driven parallel programs which can be executed on large scale and distributed environments. There have been a variety of programming models being introduced for big data with different focus and advantages. In this chapter, we will discuss and compare the major programming models for writing big data applications based on the taxonomy which is illustrated in Fig. 1.

1 MapReduce

MapReduce [24] the current defacto framework/paradigm for writing data-centric parallel applications in both industry and academia. MapReduce is inspired by the commonly used functions - Map and Reduce in combination with the divide-and-

D. Wu (✉) · S. Sakr · L. Zhu
Data61, CSIRO, Sydney, NSW, Australia
e-mail: Dongyao.Wu@data61.csiro.au

D. Wu · S. Sakr · L. Zhu
School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia

S. Sakr
King Saud Bin Abdulaziz University for Health Sciences, National Guard, Riyadh, Saudi Arabia

A.Y. Zomaya and S. Sakr (eds.), *Handbook of Big Data Technologies*,
DOI 10.1007/978-3-319-49340-4_2

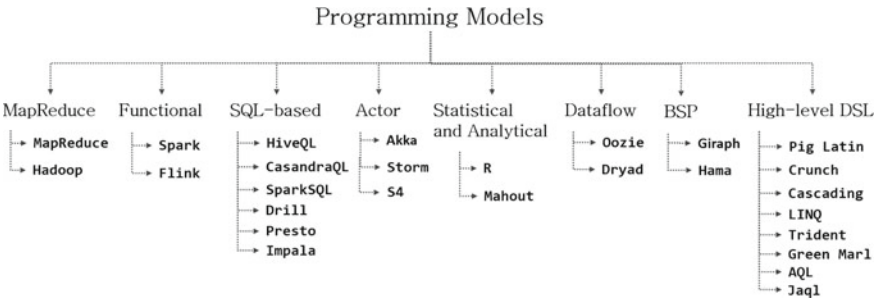

Fig. 1 Taxonomy of programming models

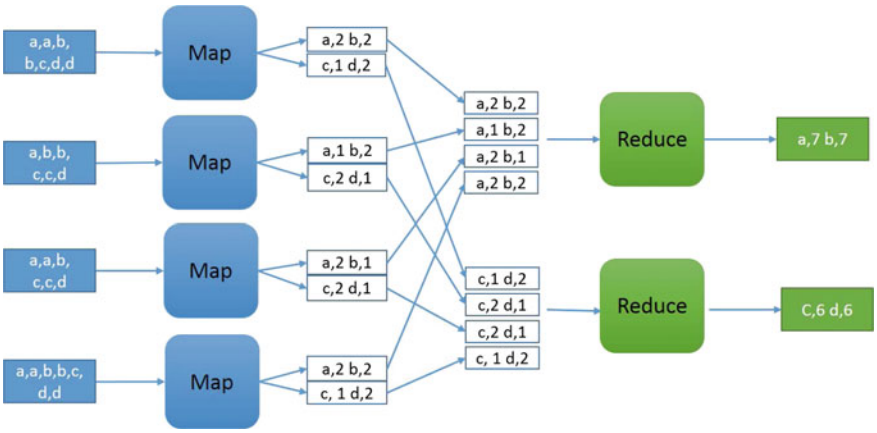

Fig. 2 MapReduce paradigm

conquer [22] parallel paradigm. For a single MapReduce job, users implement two basic procedure objects Mapper and Reducer for different processing stages as shown in Fig. 2. Then the MapReduce program is automatically interpreted by the execution engine and executed in parallel in a distributed environments. MapReduce is considered as a simple yet powerful enough programming model to support a variety of the data-intensive programs [43, 44].

1.1 Features

- *Map and Reduce functions*. A MapReduce program contains a Map function doing the parallel transformation and a Reduce function doing the parallel aggregation and summary of the job. Between Map and Reduce an implied Shuffle step is responsible for grouping and sorting the Mapped results and then feeding it into the Reduce step.
- *Simple paradigm*. In MapReduce programming, users only need to write the logic of Mapper and Reducer while the logic of shuffling, partitioning and sorting is

automatically done by the execution engine. Complex applications and algorithms can be implemented by connecting a sequence of MapReduce jobs. Due to this simple programming paradigm, it is much more convenient to write data-driven parallel applications, because users only need to consider the logic of processing data in each Mapper and Reducer without worrying about how to parallelize and coordinate the jobs.

- *Key-Value based*. In MapReduce, both input and output data are considered as Key-Value pairs with different types. This design is because of the requirements of parallelization and scalability. Key-value pairs can be easily partitioned and distributed to be processed on distributed clusters.
- *Parallelable and Scalable*. Both Map and Reduce functions are designed to facilitate parallelization, so MapReduce applications are generally linearly-scalable to thousands of nodes.

1.2 Examples

1.2.1 Hadoop

Hadoop [8] is the open-source implementation of Google's MapReduce paradigm. The native programming primitives in Hadoop are Mapper and Reducer interfaces which can be implemented by programmers with their actual logic of processing map and reduce stage transformation and processing. To support more complicated applications, users may need to chain a sequence of MapReduce jobs each of which is responsible for a processing module with well defined functionality.

Hadoop is mainly implemented in Java, therefore, the map and reduce functions are wrapped as two interfaces called Mapper and Reducer. The Mapper contains the logic of processing each key-value pair from the input. The Reducer contains the logic for processing a set of values for each key. Programmers build their MapReduce application by implementing those two interfaces and chaining them as an execution pipeline.

As an example, the program below shows the implementation of a WordCount program using Hadoop. Note that, the example only lists the code implementation of map and reduce methods but omits the signature of the classes.

Listing 1 WordCount example in Hadoop

```
1  public void map(Object key,
2                  Text value, Context context) {
3    String text = value.toString();
4    StringTokenizer itr = new StringTokenizer(text);
5    while(itr.hasMoreTokens()) {
6      word.set(itr.nextToken());
7      context.write(word, one);
8    }
9  }
10
```

```
11 public void reduce(Text key,
12         Iterable<IntWritable> values, Context context) {
13   int sum = 0;
14   for (IntWritable val : values) {
15     sum += val.get();
16   }
17   result.set(sum);
18   context.write(key, result);
19 }
```

2 Functional Programming

Functional programming is becoming the emerging paradigm for the next generation of big data processing systems, for example, recent frameworks like Spark [53], Flink [1] both utilize the functional interfaces to facilitate programmers to write data applications in a easy and declarative way. In functional programming, programming interfaces are specified as functions that applied on input data sources. The computation is treated as a calculation of functions. Functional programming itself is declarative and it avoids mutable states sharing. Compared to Object-oriented Programming it is more compact and intuitive for representing data driven transformations and applications.

2.1 Features

Functional Programming is one of the most recognized programming paradigms. It contains a set of features which facilitate the development in different aspects:

- Declarative: In functional programming, developers build the programs by specifying the semantic logic of computation rather than the control flow of the procedures.
- Functions are the first level citizens in Functional Programming. Primitives of programming are provided in functional manner and most of them can take user defined functions as parameters.
- In principle, functional programming does not allow the sharing of states, which means variables in functional programming are immutable. Therefore, there is no side effects for calling functions. This makes it easier to write functionally correct programs that are also easy to be verified formally.
- Recursive: In functional programming, many loops are normally represented as recursively calling of functions. This facilitates the optimization of performance by applying tail-recursive to reduce creating intermediate data and variables shared in different loops.
- Parallelization: As there is generally no state sharing in functional programming, it is easy and suitable for applying parallelization to multi-core and distributed computing infrastructures.

- Referential Transparent: In functional programming, as there is no states sharing and side effects. Functions are essentially re-producible. This means that re-calculation of functional results is not necessary. Therefore, once a function is calculated, its results could be cached and reused safely.

2.2 *Example Frameworks*

2.2.1 Spark

Spark provides programmers a functional programming paradigm with data-centric programming interfaces based on its built-in data model - resilient distributed dataset (RDD) [54]. Spark was developed in response to the limitations of the MapReduce paradigm, which forces distributed programs to be written in a linear and coarsely-defined dataflow as a chain of connected Mapper and Reducer tasks. In Spark, programs are represented as RDD transformation DAGs as shown in Fig. 3. Programmers are facilitated by using a rich set of high-level function primitives, actions and transformations to implement complicated algorithms in a much easier and compact way. In addition, Spark provides data centric operations such as sampling and caching to facilitate data-centric programming from different aspects.

Spark is well known for its support of rich functional transformations and actions, Table 1 shows the major transformations and operators provided in Spark. The code snippet in Listing 2 shows how to write a WoundCount program in Spark using its functional primitives.

Basically, programming primitives in Spark just look like general functional programming interfaces by hiding complex operations such as data partitioning, distribution and parallelization to programmers and leaving them to the cluster side.

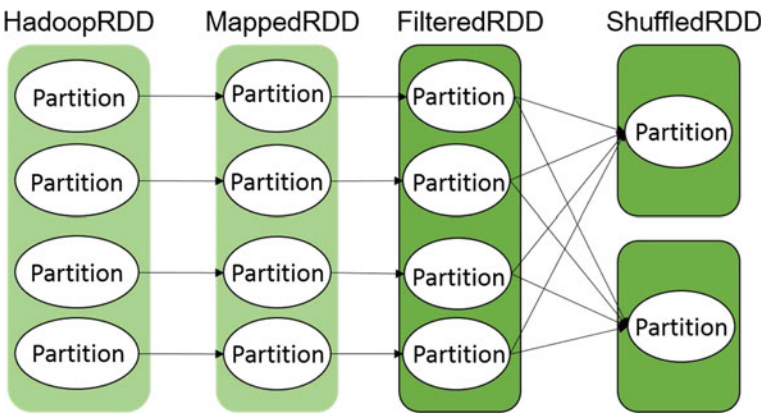

Fig. 3 RDD chain of a Spark program

Table 1 Main operations in Spark

Transformation	Description
map	Transform each element in the data set into a new type by passing them through a map function
flatMap	Transform elements in a set of collections into a plain sequence of elements by passing through a function
reduce	Aggregate elements in the data set with a function, the function needs to be commutative and associative
filter	find and return a subset of elements for the data set by checking a boolean function
groupBy	cluster the elements into different groups by mapping them with a function. The result is in (key, list of values) format
union	combine current data set with another data set to generate a new united data set
intersection	find out the overlapping elements from two data sets
distinct	find out the distinct elements from two data set
join	join two key-value based data sets and group the values for the same key. The result is in (key, (value1, value2)) format
cogroup	For two data sets of type (K, V) and (K, W), returns a grouped data set of (K, (Iterable [V], Iterable [W])) tuples

Therefore, the client side code is very declarative and simple to write by programmers. The Spark programs are essentially RDD dependency flows which will be eventually translated into parallel tasks to be executed on distributed environments.

Listing 2 WordCount example of Spark

```
1 val counts = text.flatMap{_.split("\\s+")}
2         .map{(_, 1)}
3         .reduceByKey(_ + _)
4 counts.foreach(print(_))
```

2.2.2 Flink

Flink [1] is an emerging competitor to Spark which provides functional programming interfaces that are quite similar to those of Spark. Filnk programs are regular programs which are written with a rich set of transformation operations (such as mapping, filtering, grouping, aggregating and joining) to the input data sets. Dataset in Flink is based on a table based model, therefore programmers are able to use index numbers to specify a certain field of a data set. Flink shared a lot of functional primitives and transformations in the same way as what Spark does for batch processing. The program below shows a WordCount example written in Flink.

Listing 3 WordCount example of Flink

```
val counts = text.flatMap{_.toLowerCase.split("\\s+")}
        .map{(_, 1)}
        .groupBy(0)
        .sum(1)
counts.print(_)
```

Apart from regular batch processing primitives, Flink is also natively designed for stream processing with the support of a rich set of functional operations. The streaming APIs of Flink is in its core bundle and users are able to write and execute stream processing and batch processing applications in the same framework. Unlike spark's use a min-batch to simulate stream processing, Flink uses the producer-consumer model for the execution of streaming programs. Therefore, it claims itself as being a more natural framework which integrates both batch and stream processing. Table 2 shows the major streaming operations provided in Flink.

In addition to normal transformations, the streaming API of Flink also provide a couple of window-based operations to apply functions and transformations on different groups of elements in the stream according to their time of arrival. A couple of window-based operations for Flink is listed in Table 3.

Table 2 Main operations for streaming processing in Flink

Transformation	Description
map	Takes one element and produces another one element with a transformation
flatMap	Takes one element and produces a collection of elements with a transformation function
KeyBy	Split input stream into different Partitions, each partition has the same key
reduce	Rolling combine elements in a keyed data stream with a aggregation function
filter	Retains a subset of elements from the input stream by evaluating a boolean function
fold	Same like reduce but provides a initial value and then combines the current element with the last folded value

Table 3 Window-based operations for streaming processing in Flink

Transformation	Description
window	Windows can be applied on Keyed stream to group the data in each key according to specific characteristics, e.g. arrival time
windowAll	Windows can be applied on the entire input of a stream to divide data into different groups based on specific characteristics
Window Reduce/Aggregation/Fold	After the input is grouped by windows, users can apply normal stream APIs such as reduce, fold and aggregation on the grouped streams

3 SQL-Like

SQL (Structured Query Language) is the most classic data query language, originally designed for rational databases based on the rational algebra. It contains four basic primitives: create, insert, update, delete for modifying the datasets considered as tables with schemas. SQL is a declarative language and also includes a few procedural elements.

SQL programs contain a few basic elements including: (1) Clauses which are constituent elements of statements and queries; (2) Expressions which can be valuated to produce a set of resulting data; (3) Predicates which specify conditions that can be used to limit the effects of statements and queries; (4) Queries which retrieve the data based on some specific criteria; (5) Statements which have a persistent effect on data, schema or even the database.

During execution of SQL, the SQLs are explained as syntax trees and then further translated into execution plans. And there are a bunch of optimizations have been developed to optimize the performance based on the syntax trees and execution plans.

3.1 Features

- *Declarative and self-interpretable*: SQL is a typical declarative language, it clearly specifies what transformation and operations are being done to which part of the data. By reading the SQL queries users can easily understand the semantics of the queries just like understand about literal descriptions.
- *Data-driven*: SQL is data-driven, all the operations and primitives are representing the transformation and manipulation of the target dataset (tables of data in SQL). This makes SQL one of the most facilitated programming model for data-centric applications for traditional databases and recent big data context.
- *Standardized and Inter-operable*: SQL is officially standardized by data communities such as IBM and W3C. Therefore, different platform provider can provide their own implementation while keeping the inter-operability between different platforms and frameworks. In Big data context, although there are some variations for SQL such as HQL (Hadoop Query Language) in Hive and CQL (Casandra Query Language) in Casandra, users can still easily understand and ship such programs into each other.

3.2 Examples

3.2.1 Hive Query Language (HQL)

Hive [47] is a query engine built on Hadoop ecosystems, it provides a SQL-like interface called Hive Query Language (HQL), which read input data based on defined schema and then transparently converts the queries into MapReduce jobs connected

as a directed acyclic graph (DAG). Although based on SQL, HQL does not fully follow the SQL standard. Basically, HQL lacks support for transactions and materialized views, and only support for limited indexing and sub-queries. However, HQL also provides some extensions which are not supported in SQL, such as multi-table inserts and creating table as select. The program below shows how to write a WordCount program in Hive with HQL:

Listing 4 WordCount example of Hive

```
SELECT word, count(1) AS words FROM(
SELECT EXPLODE(SPLIT(line,' ') AS word FROM myinput)
words GROUP BY word
```

Basically, HiveQL has great extent of compatibility with SQL, the Table 4 shows the semantics and data types supported in current HiveQL v0.11.

3.2.2 Cassandra Query Language (CQL)

Apache Cassandra [36] was introduced by Facebook to power up the indexing of their in-box searching. Cassandra follows the design of Amazon Dynamo with its own query interfaces - Cassandra Query Language (CQL). CQL is an SQL based query language that is provided as the alternative to the traditional RPC interface. CQL adds an abstraction layer that hides implementation details of its query structure and provides native syntaxes for collections and common encodings. The program snippet below shows some simple operations written in CQL 3.0:

Listing 5 Code example of CQL

```
BEGIN BATCH
INSERT INTO users (userID, password, name)
VALUES ('user2', 'chngemb', 'second user')
UPDATE users SET password = 'psdhds'
WHERE userID = 'user2'
INSERT INTO users (userID, password)
VALUES ('user3', 'ch@ngem3c')
DELETE name FROM users WHERE userID = 'user2'
INSERT INTO users (userID, password, name)
```

Table 4 Hive SQL compatibility

Data types	INT/TINYINT/SMALLINT/BIGINT, BOOLEAN, FLOAT, DOUBLE STRING, TIMESTAMP, BINARY, ARRAY, MAP, STRUCT, UNION DECIMAL
Semantics	SELECT, LOAD, INSERT from query, WHERE, HAVING, UNION GROUP BY, ORDER BY, SORT BY, CLUSTER BY, DISTRIBUTE BY LEFT, RIGHT and FULL INNER/OUTER JOIN, CROSS JOIN Sub-quires in FROM clause, ROLLUP and CUBE window functions (OVER, RANK, etc.)

```
10 VALUES ('user4', 'ch@ngem3c', 'Andrew')
11 APPLY BATCH;
```

Although CQL looks generally similar like SQL, there are some major differences:

- CQL uses KEYSPACE and COLUMNFAMILY compared to DATABASE and TABLE in SQL. And KEYSPACE requires more specifications (such as strategy and replication factor) than a standard relational database.
- There is no support for relation operations such as JOIN, GROUP BY, or FOREIGN KEY in CQL. Leaving these features out is important towards ensuring that the writing and retrieving data is much more efficient in Cassandra.
- Cheap writes, updates and inserts in CQL are extremely fast due to its Key-Value, and column family organization.
- Expiring records, CQL enables users to set expiry time for records by using the "USING TTL" (Time To Live) clause.
- Delayed Deletion, execution of DELETE queries doesn't really remove the data instantly. Basically, deleted records are marked with a tombstone (defined in TTL, which would exist for a period of time affected by the GC interval). Then, those marked data will be automatically removed during the upcoming compaction process.

3.2.3 Spark SQL

Spark introduces its rational query interfaces as Spark SQL [13], which is built on the DataFrame model and consider input data sets as table based structure. Spark SQL can be embedded into general programs of native Spark and MLlib [38] to enable interactability between different Spark modules. In addition, as Spark SQL draws on Catalyst to optimize the execution plans of SQL queries, Spark SQL can outperform native Spark APIs on most of the benchmarked APIs. The code snippet below shows how to define a DataFrame and use it to apply Spark SQL queries:

Listing 6 Code example of Spark SQL

```
1 val people = sc.textFile("people.txt").map(_.split(","))
2 .map(p => Person(p(0), p(1).trim.toInt)).toDF()
3 people.registerTempTable("people")
4
5 sqlContext.sql("SELECT name, age FROM people
6 WHERE age >= 13 AND age <= 19")
```

Basically, Spark SQL are embedded in the general programming context and supports most of the basic syntaxes of SQL as shown in Listing 7. Spark SQL is also compatible with various data sources including Hive, Avro [7], Parquet [10], ORC [6], JSON, JDBC and ODBC compatible databases and supports data set joins across these data sources.

Listing 7 Supported Syntax of Spark SQL

```
/* The syntax of a SELECT query */
SELECT [DISTINCT] [column names]|[wildcard]
FROM [kesypace name.]table name
[JOIN clause table name ON join condition]
[WHERE condition]
[GROUP BY column name]
[HAVING conditions]
[ORDER BY column names [ASC | DSC]]

/* The syntax of a SELECT query with joins. */
SELECT statement
FROM statement
[JOIN | INNER JOIN | LEFT JOIN | LEFT SEMI JOIN |
LEFT OUTER JOIN | RIGHT JOIN | RIGHT OUTER JOIN |
FULL JOIN | FULL OUTER JOIN]
ON join condition

/* Several select clauses can be combined in a
UNION, INTERSECT, or EXCEPT query. */
SELECT statement 1
[UNION | UNION ALL | UNION DISTINCT |
INTERSECT | EXCEPT]
SELECT statement 2

/* The syntax defines an INSERT query. */
INSERT [OVERWRITE] INTO [keyspace name.]
table name [(columns)]
VALUES values

/* The syntax defines an CACHE TABLE query. */
CACHE TABLE table name [AS table alias]

/* The syntax defines an UNCACHE TABLE query. */
UNCACHE TABLE table name
```

3.2.4 Apache Drill

Apache Drill [3] is the open source version of Google's Dremel system, which is a schema-free SQL Query Engine for MapReduce, NoSQL and Cloud Storage. Drill is well known for its connectivity to variety of NoSQL databases and file systems, including HBase [26], MongoDB [39], MapR-DB, HDFS [45], MapR-FS, Amazon S3 [46], Azure Blob Storage [15], Google Cloud Storage [27], Swift [41], NAS and

local files. A single query can join data from multiple data stores. For example, you can join a user profile collection in MongoDB with a directory of event logs in Hadoop. The main features of Apache Drill are listed as below:

- Drill uses a JSON based data model similar to MongoDB and ElasticSearch.
- Drill supports multiple industry-standard APIs, such as JDBC/ODBC, SQL and RESTful APIs.
- Drill is designed as a pluggable architecture which supports connecting with multiple data stores, including Hadoop, NoSQL and cloud-based storages.
- Drill also supports different of data formats such as JSON, Parquet and plain text.

Drill supports standard ANSI of SQL to query data from different databases and file systems regardless of its source system or its schema and data types. Listing 8 shows an example about creating a table from a JSON file in Drill.

Listing 8 Create a table from JSON data source in Drill

```
CREATE TABLE dfs.tmp.sampleparquet AS
(SELECT trans_id,
cast(`date` AS date) transdate,
cast(`time` AS time) transtime,
cast(amount AS double) amountm,
user_info, marketing_info, trans_info
FROM dfs.`/Users/drilluser/sample.json`);
```

Apart from normal SQL syntaxes, Drill also offers a couple of nested function within SQL queries as listed in Table 5.

3.2.5 Other SQL-like Query Frameworks

- Impala [21], provides high-performance, low-latency SQL queries on data stored in popular Apache Hadoop file formats. The fast response for queries enables interactive exploration and fine-tuning of analytic queries, rather than long batch jobs traditionally associated with SQL-on-Hadoop technologies. Impala integrates with the Apache Hive metastore database, to share databases and tables between both components. The high level of integration with Hive, and compatibility with the HiveQL syntax, lets you use either Impala or Hive to create tables, issue queries, load data, and so on.

Table 5 Nested functions in Drill

FLATTEN	Separate the elements in nested data from a repeated field into individual records
KVGEN	Return a repeated map generating key-value pairs for querying of complex data having unknown column names
REPEATED_COUNT	Count the values in an array
REPEATED_CONTAINS	Search for a keyword in an array

- Presto [25], is an open source distributed SQL query engine for running interactive analytic queries against data sources of all sizes ranging from gigabytes to petabytes. Presto was designed and written from the ground up for interactive analytics and approaches the speed of commercial data warehouses while scaling to the size of organizations like Facebook.

4 Actor Model

The Actor model [29] is a programming model for concurrent computation, which consider "Actor" as the universal primitive unit for computation meaning it treats everything as an actor. An actor is responsible to react to a set of messages to trigger specific processing logics (such as making decisions, building more actors, sending more messages) for different contexts. The Actor model is also considered as a reactive programming model in which programmers write acting logic in response to events and context changes. Unlike other programming models which are normally sequential, the actor model is inherently concurrent. The reactions of an actor can happen in any order and actions for different actors are also in parallel.

4.1 Features

- *Message-driven*: The Actor model inherits the message-oriented architecture for communication. messages are the primitive and the only data carrier among the systems.
- *Stateless and isolation*: Actors are loosely coupled to each other. Therefore, there is no global state shared between different actors. In addition, actors are separate functional units which are not suppose to affect others when failures and errors are encountered.
- *Concurrent*: Actors in the actor system are in action at the same time, and there is no fixed order for sending and receiving messages. Therefore, the whole actor system is inherently concurrent.

4.2 Examples

4.2.1 Akka

Akka [49] is a distributed and concurrent programming model developed by Typesafe with inspiration drawn from Erlang. Akka has been widely used in recent distributed and concurrent frameworks such as Spark, Play [50], Flink, etc. Akka provides different programming models but it emphasizes the actor-based concurrency model.

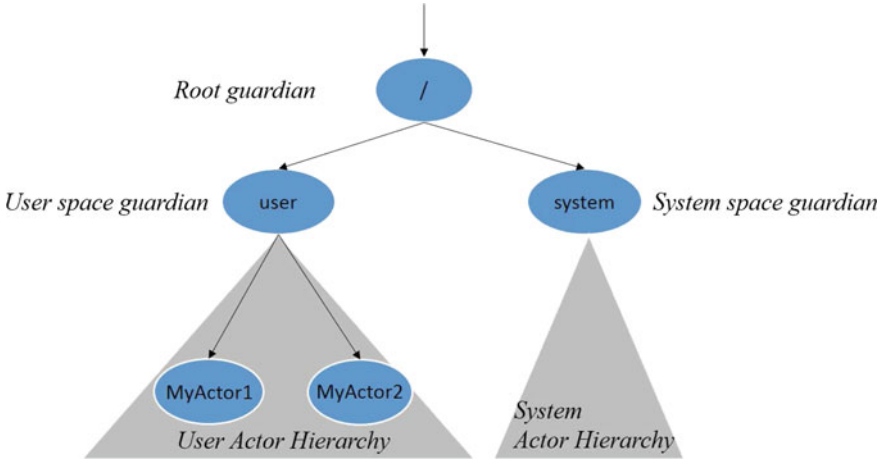

Fig. 4 Hierarchy of Akka actors

Actors in Akka communicate with each other through asynchronous messages, so typically no mutable data are shared and no synchronization primitives are used. In addition, Akka provides a hierarchical supervision model which enforces failure and fault management to the parental actors as shown in Fig. 4. Failures in Akka are also considered as messages passed to parents. Lastly, actors in Akka are portable to be executed in local and distributed environments without the need to modify existing program logics.

Akka provides a reflective way of creating a actor, in which users explicitly specify the class object and the name of an actor. In addition, Akka offers two basic message passing primitives to support communication among actors:

- ask (written as "?"), sends a message to a target actor and waits for its response as a future message.
- tell (written as "!"), sends a message to a target actor then finishes this communication, also known as fire-and-forget.

As an example, the program below shows the HelloWorld application written using Akka:

Listing 9 HelloWorld exmaple in Akka

```
class HelloActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "hello" => log.info("Hello World")
    case _       => log.info("received unknown message")
  }
}
\* create an HelloActor instance *\
val system = ActorSystem("mySystem")
```

```
val actor = system.actorOf(Props[HelloActor],"actor")
\* send a message to the actor *\
actor ! "hello"
```

4.2.2 Storm

Storm [48] is an open source programming framework for distributed realtime data processing. Storm inherits from the actor-model and provides two types of processing actors: Spouts and Bolts.

- Spout is the data source of a stream and is continuously generating or collecting new data for subsequent processing.
- Bolt is a processing entity within a streaming processing flow, each bolt is responsible for a certain processing logic, such as transformation, aggregation, partitioning and redirection, etc.

Jobs in Storm is defined as directed acyclic graphs (DAG) with connected Spouts and Bolts as vertices. Edges on the graph are data streams and direct data from one node to another. Unlike batch jobs being only executed once, Storm jobs are running until they are killed. The code snippet in Listing 10 shows an example about writing a Bolt to produce tuple streams.

In Storm, a complete application is built by connecting Spouts and Bolts. As shown in Listing 11 users can define the topology of the application by appending each Bolt to its predecessor.

Listing 10 Building a Bolt to Generate Tuple Streams

```
public class DoubleAndTripleBolt extends BaseRichBolt{
  private OutputCollectorBase _collector;

  @Override
  public void prepare(Map conf, TopologyContext context,
                      OutputCollectorBase collector){
    _collector = collector;
  }

  @Override
  public void execute(Tuple input) {
    int val = input.getInteger(0);
    _collector.emit(input, new Values(val*2, val*3));
    _collector.ack(input);
  }

  @Override
  public void declareOutputFields(
              OutputFieldsDeclarer declarer){
```

```
20      declarer.declare(new Fields("double", "triple"));
21    }
22  }
```

Listing 11 Building a WordCount Topology in Storm

```
1 TopologyBuilder builder = new TopologyBuilder();
2 builder.setSpout("words", new TestWordSpout(), 10);
3 builder.setBolt("exclaim1", new ExclamationBolt(), 3)
4         .shuffleGrouping("words");
5 builder.setBolt("exclaim2", new ExclamationBolt(), 2)
6         .shuffleGrouping("exclaim1");
```

4.2.3 Apache S4

Apache S4 (Simple Scalable Streaming System) was introduced by Yahoo in 2008 for stream processing. S4 is a general-purpose, distributed, scalable, fault-tolerant, pluggable platform for processing continuous, unbounded streams of data. The main features of S4 are listed as below:

- *Decentralized*: All nodes in S4 are symmetric with no centralized service and no single point of failure.
- *Scalable*: The throughput of S4 increases linearly as additional nodes are added to the cluster. There is no predefined limit on the number of nodes that can be supported.
- *Extensible*: Applications can be easily written using a simple API. Building blocks of the platform such as message queues and processors, serializer and checkpointing backend can be replaced by custom implementations.
- *Fault-tolerant*: Once a server in the cluster fails, a stand-by server is automatically activated to take over the tasks.

In S4, each processing actor is called a ProcessElement (PE) which is responsible to processing in response to each data element (an event) within the input stream. An S4 application is built by connecting a couple of ProcessElement in a certain topology. Listing 12 shows a example of implementing a Hello ProcessElement in S4.

Listing 12 Implement a Process Element in S4

```
1  public class HelloPE extends ProcessingElement {
2    // PEs can maintain some state
3    boolean seen = false;
4
5    // called upon every new Event on an incoming stream.
6    public void onEvent(Event event) {
7      System.out.println("Hello" + event.get("name"));
8      seen = true;
9    }
10 }
```

5 Statistical and Analytical

In recent years significant effort was spent to offer semantically friendly environments for statistical and analytical computation, which leads to the development and revolution of statistical and analytical programming models. For example many current analytics libraries or frameworks provide a linear algebra based programming model which works with vectors, matrices and tensor data structures to deal with algebraically defined mathematical problems in machine learning, statistics and data mining, etc.

5.1 Features

Due to the mathematical nature of statistical and analytical programming, it is essentially functional with manipulations on matrix and vector-based data structures.

- *Functional*: Mathematical operations are essentially functions consuming a set of input parameters to generate an output. Also many complicated functions or models are wrapped into functional libraries so that users can directly use it without knowing the implementation details of the functions.
- *Matrix-based data structure*: A matrix is one of the most widely used data structure for representing modern analytics and statistic problems and solutions. Therefore, the majority of existing analytic programming frameworks are based on matrices, vectors and data frames to manipulate the data.
- *Declarative*: In statistical and analytical programming, the programs explicitly specify the functions and operations that have been applied on the data (matrix, vector and data frame.)

5.2 Examples

5.2.1 R

R [32] combines the S [17] programming language and Lexical Scoping inspired by Scheme [20]. It is well known for statistical programming and drawing graphics. In R, data are essentially represented as matrices which are very convenient for implementing mathematical and statistical formulas. R and its libraries implement a wide variety of statistical and graphical techniques and is easy to be extended by developers. R is recently introduced to the big data processing context (RHadoop [23], SparkR [11], RHIPE [28]) to facilitate the development of statistical and analytics programs and applications. The code snippet below shows how to implement formula: $G = BB^T - C - C^T + s_q s_q^T \xi^T \xi$ using R:

Listing 13 Writing formula of R

```
g <- t(b) %*% b - c - t(c)
     + (sq %*% t(sq)) * (t(xi) %*% xi)
```

5.2.2 Mahout

Apache Mahout [9] is an open-source implementations of distributed and scalable machine learning and data mining algorithms. Mahout provides libraries that are mainly focused in the areas of collaborative filtering, clustering and classification. The initial implementation of Mahout is based on Apache Hadoop, but recently it has started to provide compatible bindings on Spark and also being able to provide matrix-based programming interfaces. For example the same formula shown in the R section can be written in Mahout as the code segment below:

Listing 14 Code example of Mahout

```
val g = bt.t %*% bt - c - c.t
        + (s_q cross s_q) * (xi dot xi)
```

The Mahout project recently (since release 1.0.0) shifts its focus to building backend-independent programming framework, which is named "Samsara". The new project consists of an algebraic optimizer and an Scala DSL to unify both distributed and in-memory algebraic operators. The current version supports execution of algebraic programs on platforms including Apache Spark and H20. The support of Apache Flink operators is also in progress.

6 Dataflow-Based

The dataflow programming paradigm models programs as directed graphs with operations and dependencies as nodes and edges. Dataflow programming [35] was first introduced by Jack Dennis and his students at MIT in the 1960s. Dataflow programming emphasizes the movement of data and considers programs as a series of connections. Every operator and processor normally has explicitly defined inputs and outputs and functions like black boxes. An operation runs as soon as all of its inputs become valid. Thus, dataflow languages are inherently parallel and can work well in large, decentralized systems.

In the big data scenario, data-centric jobs can also be modeled as dataflows in which each node represents a small task while the edges represents the data dependencies between different tasks. Developers may be need to write the process logic of each node using other general programming languages such as Java, C and Python while leave the dependency and connecting logic to the dataflow.

6.1 Features

The major features of dataflow programming can be listed as follows:

- *Trackable states*: Dataflow programming consider programs as connections of tasks in combination with control logic. Therefore, unlike other programming models, it provides a inherently trackable states during execution.
- *Various representation*: Dataflow programming model could be represented in different ways for different purposes. As we have already discussed, it can be inherently represented in a graph-based manner and also can be represented in connected texts introductions and Hash tables.

6.2 Examples

6.2.1 Oozie

Apache Oozie [34] is a server side workflow scheduling system to manage complex Hadoop jobs. In Oozie workflows are directed acyclic graphs with control flow and nodes (each node as a MapReduce jobs). In Oozie, the workflow is specified as XML-based documents presenting the connection and dataflow of different MapReduce jobs. Oozie can be integrated with other Hadoop ecosystems and also support different types of jobs such as Pig, Hive, Streaming MapReduce, etc. The XML segment below shows a Fork–Join workflow defined in Oozie:

Listing 15 Fork and Join example in Oozie

```
<workflow-app name="sample-wf"
    xmlns="uri:oozie:workflow:0.1">
    ...
    <fork name="forking">
        <path start="firstparalleljob"/>
        <path start="secondparalleljob"/>
    </fork>
    <action name="firstparallejob">
        <map-reduce>
            <job-tracker>foo:8021</job-tracker>
            <name-node>bar:8020</name-node>
            <job-xml>job1.xml</job-xml>
        </map-reduce>
        <ok to="joining"/>
        <error to="kill"/>
    </action>
    <action name="secondparalleljob">
        <map-reduce>
            <job-tracker>foo:8021</job-tracker>
```

```
20              <name-node>bar:8020</name-node>
21              <job-xml>job2.xml</job-xml>
22          </map-reduce>
23          <ok to="joining"/>
24          <error to="kill"/>
25      </action>
26      <join name="joining" to="nextaction"/>
27      ...
28  </workflow-app>
```

With the workflow specification, each action between control logic is a MapReduce associated with its Job Tracker and job definition (in a separate xml file). Actions in Oozie are triggered by time and data availability.

6.2.2 Microsoft Dryad

Microsoft Dryad [33] is a high performance, general purpose distributed computing engine which supports writing and execution of data-centric parallel programs. Dryad allows a programmer to utilize the resources of a computer cluster or a data center to run data-parallel programs. By using Dryad, programmers write simple programs which will be concurrently executed on thousands of machines (each of which with multiple processors or cores) while hiding the complexity of concurrent programming. The code below shows an example about building a graph in Dryad:

Listing 16 Building Graphs in Dryad

```
1  GraphBuilder XInputs = (ugriz1 >= XSet)
2                         || (neighbor >= XSet);
3  GraphBuilder YInputs = ugriz2 >= YSet;
4  GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
5  for (i = 0; i < N*4; ++i){
6    XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
7  }
8  GraphBuilder YToH = YSet >= HSet;
9  GraphBuilder HOutputs = HSet >= output;
10 GraphBuilder final = XInputs || YInputs
11                      || XToY || YToH || HOutputs;
```

A Dryad job contains several sequential programs and are connected using one-way channels. The program written by programmers is structured as a directed graphs, in which, programs are vertices, while the channels are edges. A Dryad job is a graph generator which can synthesize any directed acyclic graph. These graphs can also be changed during execution to respond to important events or notifications.

7 Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) [51] is a computation and programming model for designing parallel algorithms. A BSP algorithm is considered as a computation proceeds in a series of global super-steps, which consists of three components:

- *Concurrent computation*: Every participating processor may perform local computations, i.e., each process can only make use of values stored in the fast local memory of the processor. The computations occur asynchronously but may overlap with communication.
- *Communication*: The processes exchange data between themselves to facilitate remote data storage capabilities.
- *Barrier synchronisation*: When a process reaches this point (the barrier), it waits until all other processes have reached the same barrier.

7.1 Features

The BSP model contains the following key features:

- *Message-based communications*: BSP considers every communication action as a message and it also considers all messages of a super-step as a unit. This significantly reduces the effort for users to handle low-level parallel communications.
- *Barrel-based Synchronization*: BSP uses barrels to guarantee the consistency when needed, although barrel is a costly operation, it provides strong consistency and can also provides support for fault tolerance in an easy and understandable way.

7.2 Examples

7.2.1 Apache Giraph and Google Pregel

Apache Giraph [4] is an iterative graph processing system built for high scalability. Giraph is inspired by Google's Pregel [37] which is based on the Bulk Synchronous Parallel (BSP) model of distributed computation. Giraph adds several features beyond the basic Pregel model, including master computation, sharded aggregators, edge-oriented input, out-of-core computation, etc.

Listing 17 Shortest Path implementation in Giraph

```
1 public void compute(Iterable<DoubleWritable> messages){
2   double minDist = Double.MAX_VALUE;
3   for (DoubleWritable message : messages) {
4     minDist = Math.min(minDist, message.get());
5   }
```

```
    if (minDist < getValue().get()) {
      setValue(new DoubleWritable(minDist));
      for (Edge<LongWritable, FloatWritable>
           edge : getEdges()) {
        double distance = minDist+edge.getValue().get();
        sendMessage(edge.getTargetVertexId(),
                    new DoubleWritable(distance));
      }
    }
    voteToHalt();
  }
```

7.2.2 Hama

Apache Hama (stands for Hadoop Matrix) [5] is a distributed computing framework based on Bulk Synchronous Parallel computing model for massive scientific computations. Writing a Hama graph application involves inheriting the predefined Vertex class. Its template arguments define three value types, associated with vertices, edges, and messages. Hama also provides very flexible input and output options such as the ability to extract Vertex from programmers' data without any pre-processing. Hama also allows programmers to do optimizations by writing Combiner, Aggregator and Counter in data processing flows. The following code snippet shows an example of PageRank implementation in Hama:

Listing 18 PageRank implementation in Hama

```
public static class PageRankVertex extends
      Vertex<Text, NullWritable, DoubleWritable> {

 @Override
 public void compute(Iterator<DoubleWritable> messages)
                                        throws IOException {
  // initialize this vertex to 1/count
  if (this.getSuperstepCount() == 0) {
    setValue(new DoubleWritable(1.0 /
                        this.getNumVertices()));
  } else if (this.getSuperstepCount() >= 1) {
    double sum = 0;
    for (DoubleWritable msg : messages) {
      sum += msg.get();
    }
    double alpha = (1.0d-DAMPING_FACTOR)
                      / this.getNumVertices();
    setValue(new DoubleWritable(alpha +
                                  (sum*DAMPING_FACTOR)));
    aggregate(0, this.getValue());
```

```
21    }
22
23    // if have not reached global error, then proceed.
24    DoubleWritable globalError = getAggregatedValue(0);
25
26    if (globalError != null && this.getSuperstepCount()>2
27        && MAXIMUM_CONVERGENCE_ERROR>globalError.get()){
28      voteToHalt();
29    } else {
30      // in each superstep send a new rank to neighbours
31      sendMessageToNeighbors(new DoubleWritable(
32          this.getValue().get()/this.getEdges().size()));
33    }
34  }
35 }
```

8 High Level DSL

There is no a single programming model that can satisfy everyone and every scenario. Many frameworks provide their own Domain Specific Language (DSL, in contrast to general purpose programming language) for writing data-intensive parallel applications/programs in order to provide a better programming model in certain domains or purposes.

8.1 Pig Latin

Pig [40] is a high level platform to create data centric programs on top of Hadoop. The programming interface of Pig is called Pig Latin which is an ETL-like query language. In comparison to SQL, Pig uses extract, transform, load (ETL) as its basic primitives. In addition, in Pig Latin, it is able to store data at any point during a pipeline. At the same time, Pig supports the ability to declare execution plans as well as support for pipeline splits, thus allowing workflows to proceed along DAGs instead of strictly sequential pipelines. Lastly, Pig Latin scripts are automatically compiled to generate equivalent MapReduce jobs for execution.

Listing 19 WordCount example of Pig Latin

```
1 input_lines = LOAD '/tmp/wordcount-input'
2                 AS (line:chararray);
3 words = FOREACH input_lines
4         GENERATE flatten(TOKENIZE(line)) AS word;
5 filtered_words = FILTER words BY word MATCHES '\\w+';
6 word_groups = GROUP filtered_words BY word;
```

Table 6 Basic relational operators in Pig Latin

Operators	Description
LOAD	Load data from underlying file systems
FILTER	Select matched tuples from data set based on some conditions
FOREACH	Generate new data transformations based on each columns of a data set
GROUP	Group a data set based on some relations
JOIN	Join two or more data sets based on expressions of the values of their column fields
ORDERBY	Sort the data set based on one or more columns
DISTINCT	Remove duplicated elements from a given data set
MAPREDUCE	Execute native MapReduce jobs inside the Pig scripts
LIMIT	Limit the number of elements in the output

```
7  word_count = FOREACH word_groups
8               GENERATE count(filtered_words)
9               AS count, group AS word;
```

Pig offers a bunch of operators to support transformation and manipulation on input datasets. Table 6 shows the basic relational operators provided in Pig Latin and the code snippet in Listing 19 shows a WordCount example written in Pig scripts.

8.2 *Crunch/FlumeJava*

Apache Crunch [2] is high-level library supports writing testing and running data-driven pipelines on top of Hadoop and Spark. The programming interface of Crunch is partially inspired by Google's FlumeJava [19]. Crunch wraps native MapReduce interface into high level declarative primitives such as parallelDo, groupByKey, combineValues and union to make it easy for programmers to write and read their applications. Crunch provides a couple of high level processing patterns (as shown in Table 7) to facilitate developers to write data-centered applications.

Listing 20 WordCount example of Crunch

```
1  Pipeline pipeline = new MRPipeline(WordCount.class);
2  PCollection<String> lines = pipeline.readTextFile(args[0]);
3
4  DoFn<String, String> func = new DoFn<String, String>(){
5   public void process(String line,
6                       Emitter<String> emitter){
7     for (String word : line.split("\\s+")) {
8       emitter.emit(word);
9     }
10  }
```

Table 7 Common data processing patterns in Crunch

Pattern	Description
groupByKey	Group and shuffle data set based on the key of the tuples
combineValues	Aggregate elements in a grouped data set based on the combination function
aggregations	Common aggregation patterns are provided as methods on the PCollection data type, including count, max, min, and length
join	Join two keyed data sets by group the elements with the same key
sorting	Sort data set based on the value of a selected column

```
}
PCollection<String> words =
lines.parallelDo(func, Writables.strings());

for (Pair<String, Long> wordCount : words.count())) {
  System.out.println(wordCount);
}
```

In Crunch, each job is considered as a Pipeline and data are considered as Collections. Programmers write their process logic within DoFn interfaces and use basic primitives to apply transformation, filtering, aggregation and sorting to the input data sets to implement expected applications. The WordCount example of Crunch is shown in Listing 20.

8.3 Cascading

Apache Cascading [31] is a high-level development layer for building data applications on Hadoop. Cascading is designed to support the building and execution of complex data processing pipelines on a Hadoop cluster while hiding the underlying complexity of MapReduce jobs. The below code snippet shows the example of WordCount written using the Cascading API:

Listing 21 WordCount example of Cascading

```
Tap docTap = new Hfs( new AvroScheme(), docPath );
Tap wcTap = new Hfs( new TextDelimited(), wcPath, true );
Pipe wcPipe = new Pipe( "wordcount" );
wcPipe = new GroupBy( wcPipe, new Fields("count") );
wcPipe = new Every( wcPipe,
                    Fields.ALL,
                    new Count(new Fields("countcount")),
                    Fields.ALL );

FlowDef flowDef = FlowDef.flowDef()
  .setName( "wc" )
```

```
12  .addSource( wcPipe, docTap )
13  .addTailSink( wcPipe, wcTap );
14
15 Flow wcFlow = flowConnector.connect( flowDef );
16 wcFlow.writeDOT( "dot/wcr.dot" );
17 wcFlow.complete();
```

As we can see from the example, a cascading job is defined as a Flow, in which it can contains multiple pipes. Each pipe is actually a function block which is responsible for a certain data process step such as GroupBy, Filtering, Joining and Sorting. Pipes are connected to construct the final Flow for execution.

8.4 Dryad LINQ

DryadLINQ [52] is a compiler which translates LINQ (Language-Integrated Query) programs to distributed computations which can be run on a cluster. The goal of LINQ is to bridge the gap between the world of objects and the world of data. LINQ uses query expressions akin to SQL statements such as select, where, join, groupBy and orderBy, etc. In addition, LINQ also defines a set of method names (called standard query operators), along with translation rules used by the compiler to translate fluent-style query expressions into expressions using these method names, lambda expressions and anonymous types. In DryadLINQ, the data queries are automatically compiled as DAG tasks to be executed on the Dryad engine to support the building and execution of large scale data-driven applications and programs.

Listing 22 WordCount example of Dryad LINQ

```
1  public static IQueryable<Pair> Histogram(
2                IQueryable<string> input, int k){
3   IQueryable<string> words =
4              input.SelectMany(x => x.Split(' '));
5   IQueryable<IGrouping<string, string>> groups =
6              words.GroupBy(x => x);
7   IQueryable<Pair> counts =
8              groups.Select(x => new Pair(x.Key, x.Count()));
9   IQueryable<Pair> ordered =
10             counts.OrderByDescending(x => x.count);
11  IQueryable<Pair> top = ordered.Take(k);
12  return top;
13 }
```

The code snippet above shows an example of the WordCount program written using DryadLINQ. As we can see from the example, LINQ actually provides ETL operations in an Object-oriented ways. Query primitives are object operations which associated to the data, and the result of queries are represented as collections with specific types.

8.5 Trident

Trident [12] is a high-level abstraction for doing realtime computing on top of Storm. It allows you to seamlessly intermix high throughput (millions of messages per second), stateful stream processing with low latency distributed querying. If you're familiar with high level batch processing tools like Pig or Cascading, the concepts of Trident will be very familiar. Trident has joins, aggregations, grouping, functions, and filters. In addition to these, Trident adds primitives for doing stateful, incremental processing on top of any database or persistence store. Trident has consistent, exactly-once semantics, so it is easy to reason about Trident topologies.

Listing 23 Code snippet of WordCount using Trident

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
      topology.newStream("spout1", spout)
      .each(new Fields("sentence"),
            new Split(),
            new Fields("word"))
      .groupBy(new Fields("word"))
      .persistentAggregate(new MemoryMapState.Factory(),
                           new Count(),
                           new Fields("count"))
      .parallelismHint(6);
```

8.6 Green Marl

Green Marl [30] is a DSL introduced by the Pervasive Parallelism Laboratory of Stanford University and specifically designed for graph analysis. Green Marl allows user to describe their graphs intuitively through a high level interface while inherently provide data-driven parallelism. Green Marl, provides the ability to define both directed graphs and undirected graphs and supports basic types (like Int, Long, Float, Double and Bool) and collections (like Set, Sequence and Order). Green Marl introduces its own compiler to interpret the program into C++ code for execution. The compiler of Green Marl also introduces a couple of optimizations during compile-time to improve the execution performance. The code snippet below shows an example about the Betweenness Centrality algorithm written in Green Marl.

Listing 24 Betweenness Centrality algorithm described in Green Marl

```
Procedure Compute_BC(
 G: Graph, BC: Node_Prop<Float>(G)) {
 G.BC = 0; // initialize BC
 Foreach(s: G.Nodes) {
 // define temporary properties
```

```
6  Node_Prop<Float>(G) Sigma;
7  Node_Prop<Float>(G) Delta;
8  s.Sigma = 1; // Initialize Sigma for root
9  // Traverse graph in BFS-order from s
10 InBFS(v: G.Nodes From s)(v!=s) {
11 // sum over BFS-parents
12 v.Sigma = Sum(w: v.UpNbrs) {w.Sigma};
13 }
14 // Traverse graph in reverse BFS-order
15 InRBFS(v!=s) {
16 // sum over BFS-children
17 v.Delta = Sum (w:v.DownNbrs) {
18 v.Sigma / w.Sigma * (1+ w.Delta)
19 };
20 v.BC += v.Delta; //accumulate BC
21 } } }
```

8.7 Asterix Query Language (AQL)

The Asterix Query Language (AQL) [14] is the language interface provided by AsterixDB which is a scalable big data management system (BDMS) with the capability for querying semi-structured data sets. AQL is based on a NoSQL style data model (ADM) which extends JSON with object database concepts. Basically, AQL is an expressive and declarative query language for querying semi-structured data with the support for a rich set of primitive types, including spatial, temporal and textual data. The code snippet below shows an example about joining two data sets in AQL.

Listing 25 Join two data sets by AQL

```
1 for $user in dataset FacebookUsers
2 for $message in dataset FacebookMessages
3 where $message.author-id = $user.id
4 return
5   {
6     "uname": $user.name,
7     "message": $message.message
8   };
```

As we can see from the example code, AQL combines the style of an SQL query with the data model of JSON to provide a programming style that is both declarative and data-driven. The core of AQL is called FLWOR (for-let-where-orderby-return) expression which is borrowed from XQuery expressions. A FLWOR expression starts with one or more clauses which establishes the variable bindings.

- A *for* clause binds a variable incrementally to each element of its associated expression and includes an optional positional variable for counting/numbering the bindings.
- A *let* clause binds a variable to the collection of elements computed by its associated expression.
- The *where* clause in a FLWOR expression filters the preceding bindings via a boolean expression, much like a where clause does in an SQL query.
- The *order by* clause in a FLWOR expression induces an ordering on the data.
- The *return* clause defines the data expected to be sent back as the results of a query.

8.8 IBM Jaql

Jaql (or JAQL) [18] is a functional data processing and query language mostly focusing on JSON-based query processing on BigData. Jaql was originally introduced by Google and then further developed by IBM. Jaql is designed to elegantly handle deeply nested semi-structured data and even deal with heterogeneous data. Jaql can also be used in Hadoop as a expressive query language that is comparable with Pig and Hive. The code snippet below shows some basic examples of queries written in Jaql.

Listing 26 Basic opertions in Jaql

```
1  a = {name : "scott", age : 42, children : ["jake", "sam"]};
2  a.name; //returns "scott"
3  a.children[0]; // returns "jake"
4
5  // for local file system
6  read(del("file:///home/user/test.csv"));
7  // for hdfs file system
8  read(del("hdfs://localhost:9000/user/test.csv"));
9
10 recs = [ {a: 1, b: 4}, {a: 2, b: 5}, {a: -1, b: 4} ];
11 recs -> transform .a+.b; // returns [ 5, 7, 4 ]
```

9 Discussion and Conclusion

In this section, we summarize the main features and compare the various programming models presented in this chapter.

Due to its declarative feature, functional programming is natural fit for data-driven programs and applications. As pure functions are stateless and have no side effects, functional programs are easier to be parallelized and proofed for correctness. In addition, functional programs are easier to debug and test as functions are a better isolation of functionalities without the uncertainties caused by state sharing and other

side-effects. However, programming in a functional way is much different from programming in the imperative programming. Developers need to shift from imperative and procedure-based thinking to a functional way of thinking when writing the programs. This may require considerable efforts from the developer to learn and practice in order to gain sufficient mastery.

MapReduce is considered as an easy way of writing data-driven parallel programs. The emergence of MapReduce significantly eased the task for developing data-parallel applications on large scale data sets. Although the paradigm is simple, it can still cover the majority of the algorithms in practice. MapReduce is not guaranteed to be fast as its focus is more on scalability and fault tolerance. In addition, MapReduce is criticized for lacking the novelty of more recent developments and its restricted programming paradigm which does not support iterative and streaming algorithms.

SQL is considered as having limited semantics and not sufficiently expressive. Basically, SQL is not a Turing-complete language, it is more towards a query rather than a general programming language such as Java and C. As a result, it is more suitable for writing ETL (Extract, Transform and Load) or CRUD (Create, Read, Update and Delete) queries rather than general algorithms. For example, it would be a horrible choice to use SQL for writing data mining and machine learning algorithms. Traditional rational queries are slow and less scalable in Big Data scenarios, therefore, query languages such as HQL and CQL cut down the majority of the rational paradigms provided by traditional SQL in order to be more scalable in a big data context.

The first advantage of dataflow programming is that it facilitates for visualized programming and monitoring. Due to its simplified graph-based interfaces, it is easy to prototype and implement certain systems and applications. In addition, it is

Table 8 Comparison for different programming models

Model	Features	Abstraction	Semantics	Computation model
MapReduce	Non-declarative skeleton-based	low	Limited inherent parallel	MapReduce
Functional	Declarative stateless	High	Rich and general purpose	DAG or Evaluation-based
SQL-based	Declarative	High	Limited	Execution plan
Data flow	Non-declarative modularized	mediate	Rich control-logic based	DAG
Statistical	Declarative	High	Limited domain-specific	Mathemetical
BSP	Skeleton-based	Low	Rich	BSP
Actor	Event-driven message-based	Low	Rich and inherent concurrency	Reactive actors

also well known for providing end-user programming in which WYSIWYG (what you see is what you get) interfaces are required. Another point in favour of data flow programming is that, by writing programs in a dataflow manner, it actually help developers to modularize their programs as connected processing components and provide good loosely coupled structure and flexibility. However, compared to other programming models such as functions and SQLs, dataflow-based model is relatively non-declarative, unproductive for programming as it basically provides low-level programming abstractions and interfaces, which is hard to be integrated with.

To sum up, the comparison of basic programming models are listed in Table 8. Basically, MapReduce and BSP models are programming skeletons, functional, SQL and statistical models are declarative, while data-flow model is inherently modularized and actor model is essentially event-driven and message-based. In addition, functional, SQL and statistical models are high-level abstraction models and MapReduce, BSP and actor models are low-level abstraction. Lastly, functional, BSP and actor models are more semantically complete to support richer operations while other models are generally limited in semantics to trade off among understandability, user-convenience and productivity [16, 42].

References

1. A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M.J. Sax, S. Schelter, M. Höger, K. Tzoumas, D. Warneke, The stratosphere platform for big data analytics, VLDB J. **23**(6) (2014)
2. Apache. Apache crunch (2016). https://crunch.apache.org/. Accessed 17 Mar 2016
3. Apache. Apache drill (2016). https://drill.apache.org/. Accessed 17 Mar 2016
4. Apache. Apache giraph (2016). https://giraph.apache.org/. Accessed 17 Mar 2016
5. Apache. Apache hama (2016). https://hama.apache.org/. Accessed 17 Mar 2016
6. Apache. Apache orc (2016). https://orc.apache.org/. Accessed 17 Mar 2016
7. Apache. Avro (2016). https://avro.apache.org/. Accessed 17 Mar 2016
8. Apache. Hadoop (2016). http://hadoop.apache.org/. Accessed 17 Mar 2016
9. Apache. Mahout: Scalable machine learning and data mining (2016). https://mahout.apache.org/. Accessed 17 Mar 2016
10. Apache. Parquet (2016). https://parquet.apache.org/. Accessed 17 Mar 2016
11. Apache. Spark r (2016). https://spark.apache.org/docs/1.6.0/sparkr.html. Accessed 17 Mar 2016
12. Apache Storm. Trident (2016). http://storm.apache.org/documentation/Trident-tutorial.html. Accessed 17 Mar 2016
13. M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, M. Zaharia, Spark SQL: relational data processing in spark, in *SIGMOD* (2015), pp. 1383–1394
14. AsterixDB. Asterix query language (aql) (2016). https://asterixdb.ics.uci.edu/documentation/aql/manual.html. Accessed 17 Mar 2016
15. Azure Microsoft. Microsoft azure: Cloud computing platform and services (2016). https://azure.microsoft.com. Accessed 27 Feb 2016

16. O. Batarfi, R. El Shawi, A.G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, S. Sakr, Large scale graph processing systems: survey and an experimental evaluation. Clust. Comput. **18**(3), 1189–1213 (2015)
17. R.A. Becker, J.M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics* (CRC Press, New York, 1984)
18. K.S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, E.J. Shekita, Jaql: a scripting language for large scale semistructured data analysis, in *Proceedings of VLDB Conference* (2011)
19. C. Chambers, A. Raniwala, F. Perry, S. Adams, R.R. Henry, R. Bradshaw, N. Weizenbaum, FlumeJava: easy, efficient data-parallel pipelines, in *PLDI* (2010)
20. W. Clinger, J. Rees, Ieee standard for the scheme programming language, in *Institute for Electrical and Electronic Engineers* (1991), pp. 1178–1990
21. Cloudera. Apache impala (2016). http://impala.io/. Accessed 17 Mar 2016
22. T.H. Cormen, *Introduction to Algorithms* (MIT press, New York, 2009)
23. S. Das, Y. Sismanis, K.S. Beyer, R. Gemulla, P.J. Haas, J. McPherson, Ricardo: integrating r and hadoop, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (ACM, 2010), pp. 987–998
24. J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1) (2008)
25. Facebook. Presto (2016), https://prestodb.io/. Accessed 17 Mar 2016
26. L. George, *HBase: The Definitive Guide* (O'Reilly Media, Inc., 2011)
27. Google. Cloud sql - mysql relational database (2016). https://cloud.google.com/sql/. Accessed 27 Feb 2016
28. S. Guha, R. Hafen, J. Rounds, J. Xia, J. Li, B. Xi, W.S. Cleveland, Large complex data: divide and recombine (d&r) with rhipe. Stat **1**(1), 53–67 (2012)
29. C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Morgan Kaufmann Publishers Inc., 1973), pp. 235–245
30. S. Hong, H. Chafi, E. Sedlar, K. Olukotun, Green-marl: a dsl for easy and efficient graph analysis, in *ACM SIGARCH Computer Architecture News*, vol. 40 (ACM, 2012), pp. 349–362
31. Inc Concurrent. Cascading - application platform for enterprise big data (2016). http://www.cascading.org/. Accessed 17 Mar 2016
32. R. Ihaka, R. Gentleman, R: a language for data analysis and graphics. J. Comput. Graph. Stat. **5**(3), 299–314 (1996)
33. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in *ACM SIGOPS Operating Systems Review*, vol. 41 (ACM, 2007), pp. 59–72
34. M. Islam, A.K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, A. Abdelnur, Oozie: towards a scalable workflow management system for hadoop, in *SIGMOD Workshops* (2012)
35. W.M. Johnston, J.R. Hanna, R.J. Millar, Advances in dataflow programming languages. ACM Comput. Surv. (CSUR) **36**(1), 1–34 (2004)
36. A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
37. G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in *SIGMOD Conference* (2010)
38. X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D.B. Tsai, M. Amde, S. Owen, et al., Mllib: machine learning in apache spark (2015). arXiv preprint, arXiv:1505.06807
39. MongoDB Inc. Mongodb for giant ideas (2016). https://www.mongodb.org/. Accessed 27 Feb 2016
40. C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in *SIGMOD* (2008)

41. Swift OpenStack. Openstack swift - enterprise storage from swiftstack (2016). https://www.swiftstack.com/openstack-swift/. Accessed 27 Feb 2016
42. S. Sakr, *Big Data 2.0 Processing Systems* (Springer, Berlin, 2016)
43. S. Sakr, M.M. Gaber (eds.) *Large Scale and Big Data - Processing and Management* (Auerbach Publications, 2014)
44. Sherif Sakr, Anna Liu, Ayman G. Fayoumi, The family of mapreduce and large-scale data processing systems. ACM Comput. Surv. **46**(1), 11 (2013)
45. K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in *IEEE MSST* (2010)
46. S3 Amazon. Amazon simple storage service (amazon s3) (2016). https://aws.amazon.com/s3/. Accessed 27 Feb 2016
47. A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
48. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al., Storm@ twitter, in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (ACM, 2014), pp. 147–156
49. Typesafe. Akka (2016). http://akka.io/. Accessed 17 Mar 2016
50. Typesafe. Play framework - build modern & scalable web apps with java and scala (2016). https://www.playframework.com/. Accessed 17 Mar 2016
51. L.G. Valiant, A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
52. Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, J. Currey, Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language, in *OSDI*, vol. 8 (2008), pp. 1–14
53. M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in *HotCloud* (2010)
54. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in *NSDI* (2012)

http://www.springer.com/978-3-319-49339-8

Handbook of Big Data Technologies
Zomaya, A.Y.; Sakr, S. (Eds.)
2017, XIII, 895 p. 307 illus., Hardcover
ISBN: 978-3-319-49339-8