COMP5349 – Cloud Computing

Week 7: Data Flow Engines for Cloud Analytics

A/Prof Dr Uwe Röhm
School of Information Technologies

Friday, 1.30pm - 2.30pm
Friday, 2pm - 3pm

Lecture Outlook

- Today:
  - Help with Assignment 1 in the lab rooms in SIT
  - Submission by Friday (tomorrow) 6pm in Blackboard
  - Please also fill in the **Self-Reflection Survey after** finishing the assignment

- Next Week: Details on Spark and Flink
  - Including lab
  - A2 to be published in Week 8

- Friday this week (28th April): Data Centre Excursion
  - Visit to one of the Equinix data centres in Alexandria (200 Bourke Rd)
  - 2 tours @ 1-1:30pm and @ 1:45pm-2:30pm

  => ENROL IN TOUR GROUPS IN Blackboard (under 'Groups')

Equinix Field Trip

- Tomorrow, Friday 28th April: Data Centre Excursion
  - 2 tours @ 1-1:30pm and @ 1:45pm-2:30pm
  - First-Come-First-Served
  - **ENROL to TOUR GROUPS IN Blackboard ('Groups' section)**
- Requirements
  - 18+ and an **official Australian photo ID** (drivers license; passport)
    - International students: need to bring passport
    - Note: your student ID card is not accepted
  - Closed-in shoes
  - Signed Fieldtrip Acknowledgement Form
- Location: 200 Bourke Road, Alexandria
  - Public Trapo: 10 minutes walk from Mascot Train Station
  - Car… Limited parking at Bunnings which is next door

Outline

- **Motivation**
- Overview of Apache Spark
- Overview of Apache Flink
- Conclusions

Based on slide decks from Ion Stoica, Mike Franklin,
as well as the online documentation from BDAS, Stratosphere and Flink.

Original Cloud Analytics Stack

- … mostly focused on large, on-disk datasets: great for ***batch***, but ***slow***

Application

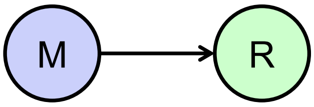
Data Processing

Storage

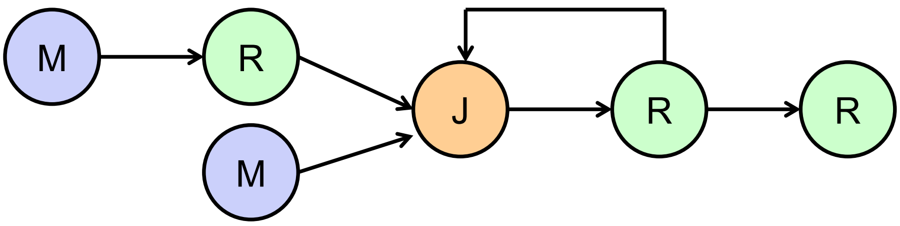
Infrastructure

Motivation

- Map/Reduce allows simple parallelization of pipelined tasks
  - Pro: parallelism, fault-tolerance, runtime can decide where to run tasks
  - Con: simple processing model that assumes data flowing from stable storage to stable storage + materialization of intermediate results



- Can't we do better?



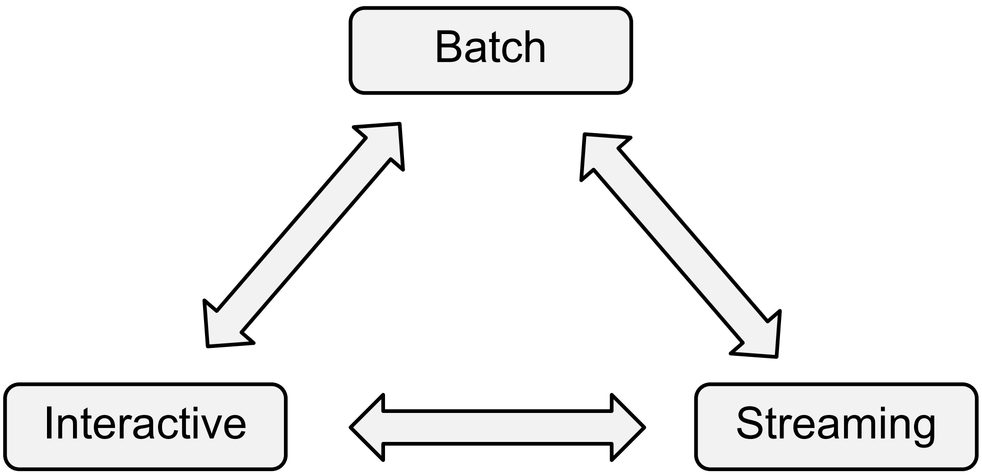
Hive / Pig?

- There are several approaches to provide SQL-like query languages on top of map reduce

- Pro:
  - High-level abstraction of execution
  - Data centric view
  - SQL

- Con:
  - Core execution model still Map/Reduce jobs
  - Add significant execution overhead
  - Expressiveness of SQL is limited; e.g. no machine learning or graph processing possible

Goals



- How to combine ***batch***, ***streaming***, and ***interactive*** computations?
- Need for support for more sophisticated algorithms, and easy of use.
- Ideally: compatible with existing infrastructure (Hadoop/HDFS)

Approach

- Extensive use of main ***memory*** for processing
  - Rationale: memory is cheap nowadays and much faster than disk (even than SSDs)
  - Many datasets already fit into memory
    - The inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory [cite]
- Acyclic data flow plan with advanced operators
  - Go beyond simple map/reduce for more expressive tasks
- Increased parallelism
  - Ideally via automated plan optimisation and scheduling
- Built on top of Hadoop/HDFS
  - Usable with existing jobs and data stores

Berkeley Data Analytics Stack (BDAS)

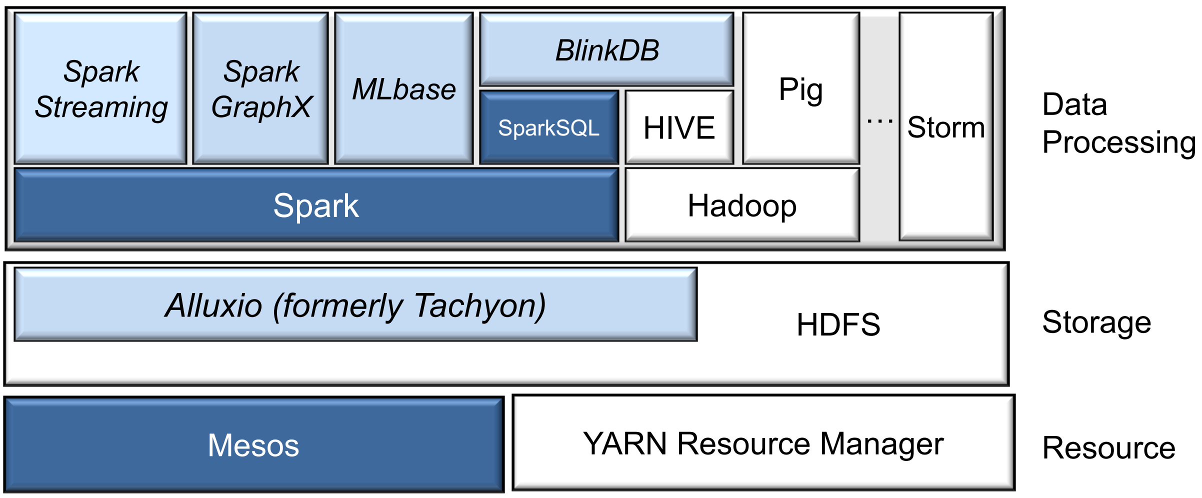
https://amplab.cs.berkeley.edu/bdas/

BDAS Approach

- Project by Amplab, UC Berkeley
- Single execution model that supports *batch*, *streaming*, and *interactive* computation models
- Easy to develop sophisticated algorithms
  - Powerful Python and Scala shells
  - High level abstractions for graph based, and ML algorithms
- Compatible with existing open source ecosystem (Hadoop/HDFS)
  - Interoperate with existing storage and input formats (e.g., HDFS, Hive, Flume, ..)
  - Support existing execution models (e.g., Hive, GraphLab)

[slide by Ion Stoica, UCB, 2013]

Berkeley Data Analytics Stack



- Several extensions based on Hadoop/HDFS
- **Mesos**: Multi-tenant Resource Management
  - Management platform that allows multiple framework to share cluster; used by eg. Twitter and airbnb
- **Spark**: In-memory framework for interactive and iterative computations
  - Scala interface, Java and Python APIs
- **SparkSQL**: "HIVE over Spark" - a SQL-like interface, compatible to HIVE queries
- Many more in alpha: Alluxio (in-memory storage), Streaming, MLBase, BlinkDB, …

[graph by Ion Stoica, UCB, 2013]

Apache Spark

- In-memory framework for interactive and iterative computations
- Goals:
  - distributed memory abstractions for clusters to support apps with working sets
  - Retain the attractive properties of MapReduce:
    - Fault tolerance (for crashes & stragglers)
    - Data locality
    - Scalability
- Approach:
  - augment data flow model with **Resilient Distributed Dataset (RDD)**
    - RDD: fault-tolerance, in-memory storage abstraction

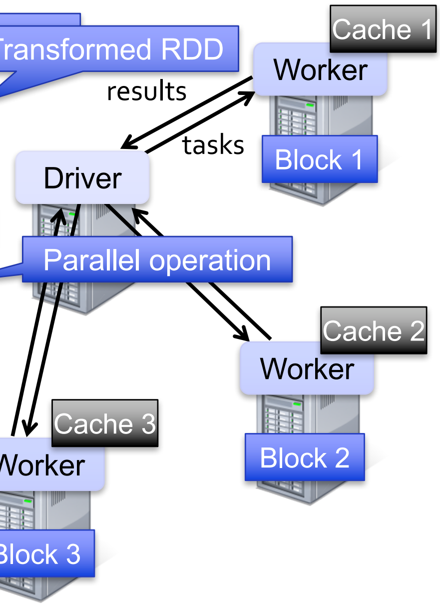
Spark Programming Model

- **Resilient distributed datasets (RDDs)**
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - Reduce, collect, count, save, …
- Restricted shared variables
  - Accumulators, broadcast variables

Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

RDDs in More Detail

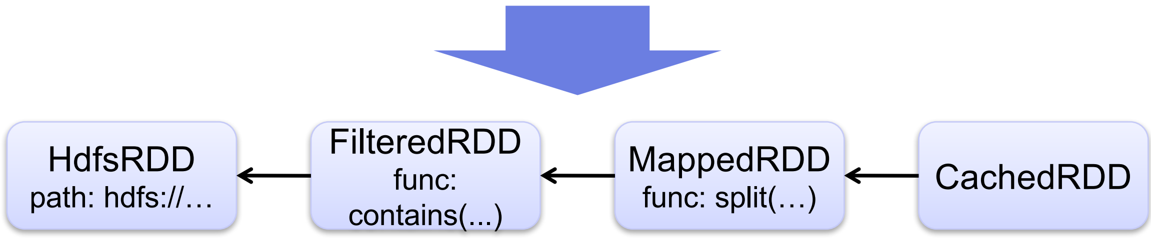
- RDD: immutable (read-only), partitioned, logical collection of records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record
  - hash or range partitioning
- Built using bulk transformations on other RDDs
- Can be cached for future reuse
- Operations:

Transformations (define a new RDD)	**Parallel operations** (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

RDD Fault Tolerance

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions
- Example:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```

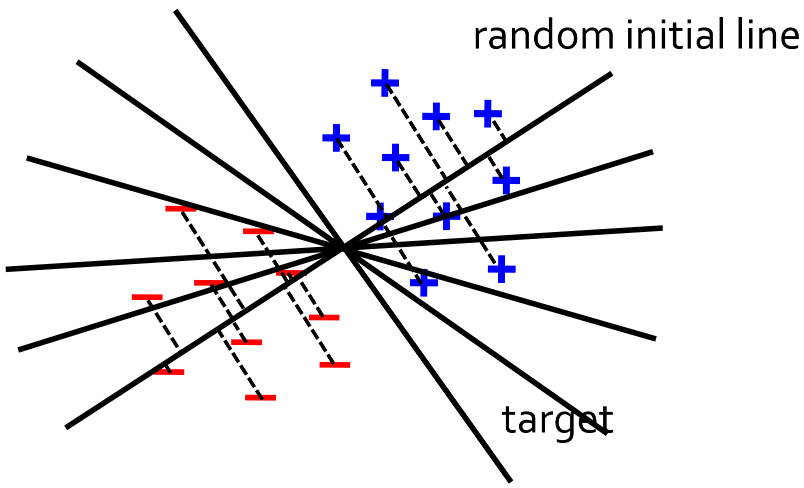


Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

Example App: Logistic Regression

- Goal: find best line separating two sets of points



Example Logistic Regression Code

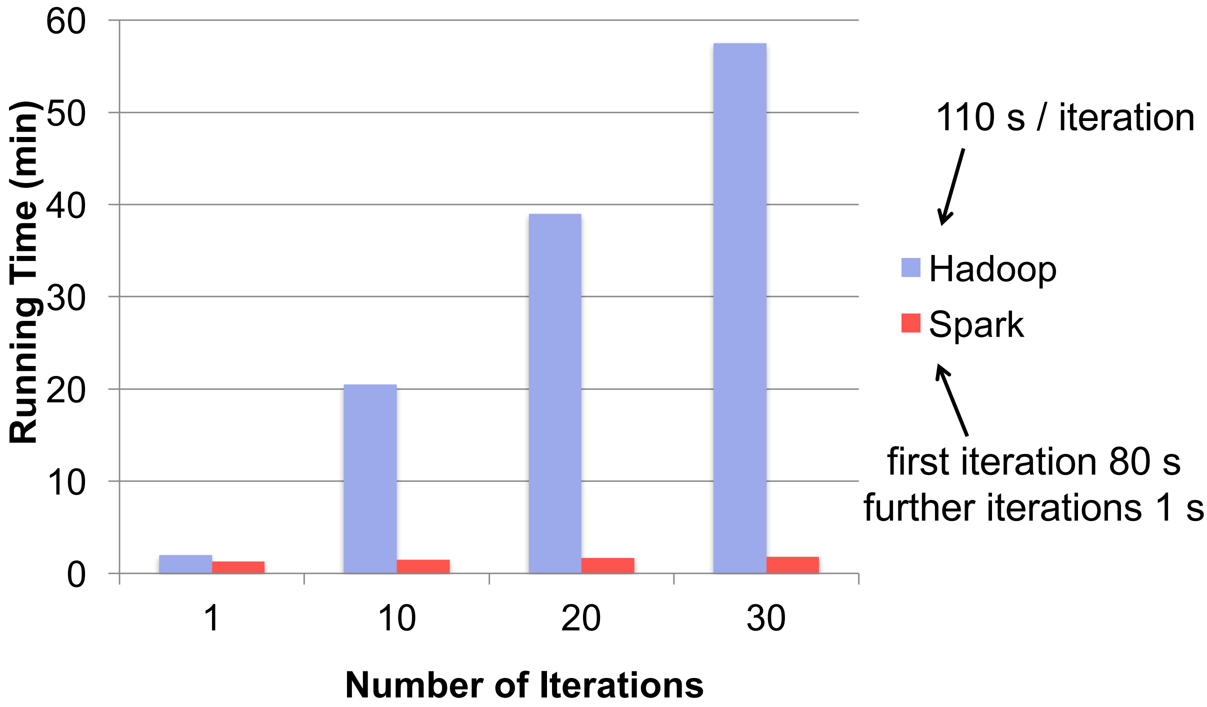
Load data in memory once

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Initial parameter vector

Repeated MapReduce steps to do gradient descent

Logistic Regression Performance



29 GB dataset on 20 EC2 m1.xlarge machines (4 cores each)

Example: MapReduce

- MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

Word Count in Spark

```
val lines = spark.textFile("hdfs://...")

val counts = lines.flatMap(_.split("\\s"))
                  .reduceByKey(_ + _)

counts.save("hdfs://...")
```


BDAS: Work in Progress

- **Tachyon** [alpha release so far]
  - High-throughput, fault-tolerant in-memory storage
  - Interface compatible to HDFS; support for Spark and Hadoop
- **BlinkDB** [alpha release]
  - large scale approximate query engine
  - Allow users to specify error or time bounds
- **SparkGraph** [alpha release]
  - GraphLab API and Toolkits on top of Spark
  - Fault tolerance by leveraging Spark
- **MLbase** [in development]
  - Declarative approach to ML
  - Develop scalable ML algorithms
  - Make ML accessible to non-experts

Apache Flink

http://flink.apache.org

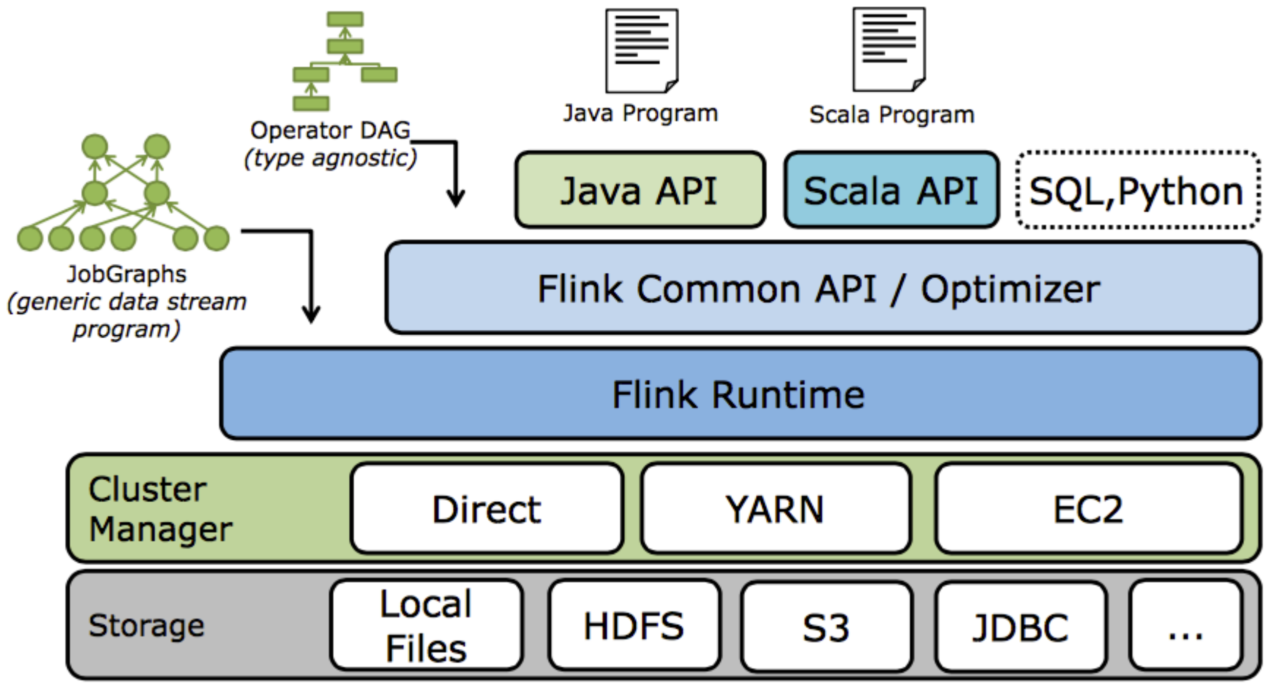
Apache Flink Project

- Started as a research project ('Stratosphere') by database research groups at TU Berlin, Humboldt University Berlin and HPI Potsdam
- Since Feb 2015 Apache top-level project (Apache Flink)
- Efficient data flow runtime on top of Hadoop/HDFS/YARN
  - Similar scalability and fail safety
  - More powerful data flow operators and optimization component
- Seamlessly integrates into existing Hadoop infrastructure:
  - can run side-by-side with Hadoop's TaskTrackers and DataNodes
  - can read data from Hadoop sources

Core Features

- Enhanced Execution Engine
  - Multiple data transformation:
    - Join, Cross, Union, Iterate, …
  - In-memory pipelining between operators
  - Support for Iterative Algorithms
  - Lazy Evaluation
- Built-In Optimizer
  - Cost-based optimizer: execution strategy depending on inputs & ops
  - For example: "Join" operator
    - Sort-merge-join, hash-join, broadcasting…
  - Input Sampling to determine cardinalities
- Enhanced APIs
  - Support for Java and Scala

Flink System Stack



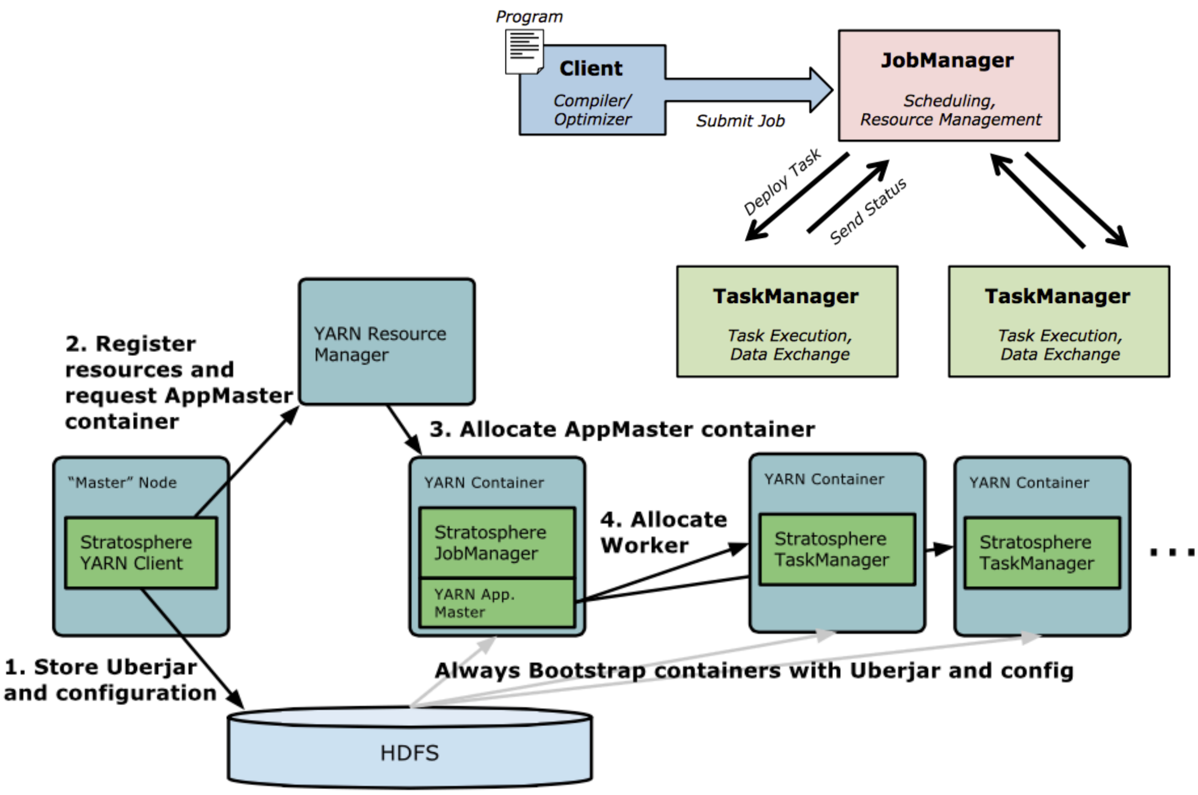
Source: http://ci.apache.org/projects/flink/flink-docs-release-0.8.1/internal_general_arch.html

Example: Scala API

- **Word count** in Flink using Scala:

```
val input = TextFile(textInput)

val words = input.flatMap { line => line.split(" ") }
val counts = words
  .map { (_, 1) }
  .groupBy (0)
  .sum(1)

counts.print();

env.execute("Scala WordCount Example")
```


Flink on Yarn

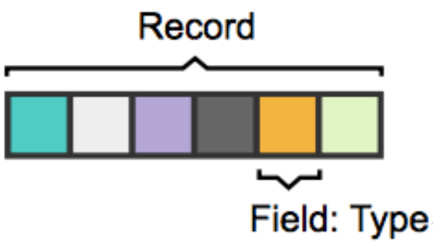


Flink Programming Model

- Complex data analysis tasks specified as a data flow graph between parallelizable operators
  - Inherits aspects from MapReduce, but goes beyond it
- Aspects:
  - Data Model
  - Data Transformations
  - Sequential Data Flows
  - Iterative Data Flows

Flink: Data Model

- Inherent data model of Flink: **Tuple** (or record)
  - A tuple consists of an arbitrary number of (nested) data fields
  - Fields can be of a primitive data type, or a nested tuple structure



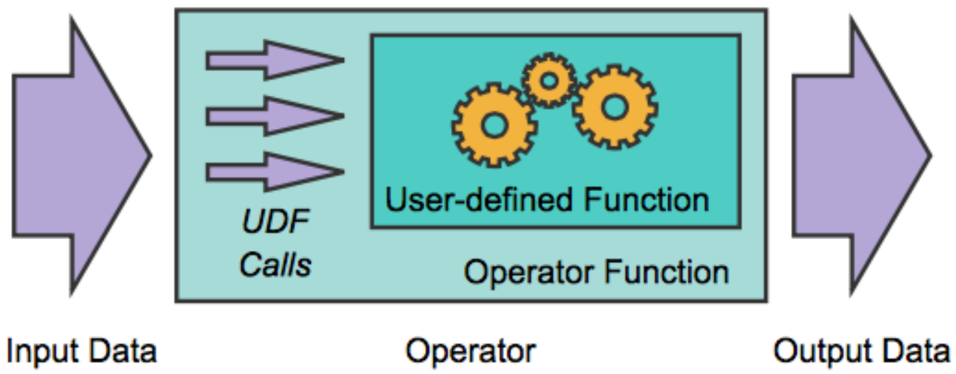
- General (primitive) Data Types (predefined)
  - **Integer values**: Byte, Short, Integer, Long
  - **Floating point values**: Float, Double
  - **Text values**: Character, String
  - **Special values**: Boolean, Null

Flink Data Model (cont'd)

- Custom (User-defined) Data Types supported too
- ***Value*** types describe their serialization and deserialization manually
  - custom code for those operations can be implemented by means of the *org.apache.flinktypes.Value* interface with the methods *read* and *write*
  - Flink comes with pre-defined Value types that correspond to basic data types, and that in contrast to the basic data types are mutable. (ByteValue, ShortValue, IntValue, LongValue, FloatValue, DoubleValue, StringValue, CharValue, BooleanValue)
- A collection of tuples/records is called a **DataSet**
  - Operators receive one or more data sets as input and produce one new data set

Flink: Data Transformations

- programming model based on parallelizable operators
- An operator consists for two components
  - *user-defined function (UDF)* and
  - a parallel *operator function*.
    - The operator function parallelizes the execution of the user-defined function and applies the UDF on its input data.



Data Transformation (cont'd)

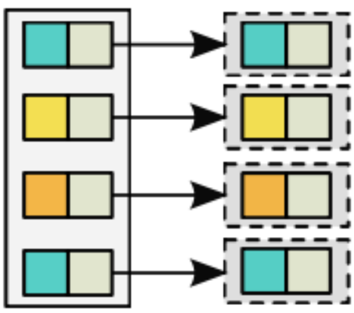
- Flink's programming model provides several parallelizable data transformations:
  - *Map*,
  - *Reduce* (including an optional *Combine*),
  - *Filter,*
  - *Aggregate,*
  - *Join*,
  - *Cross*,
  - *CoGroup*,
  - *Union*
  - *…*

[http://ci.apache.org/projects/flink/flink-docs-release-0.8/dataset_transformations.html]

- Map, Filter and Reduce operate on a single input
- Join, Cross, and CoGroup combine the data of two input
- Map, Filter, Join, and Cross operate on individual records, Reduce and CoGroup on groups of records

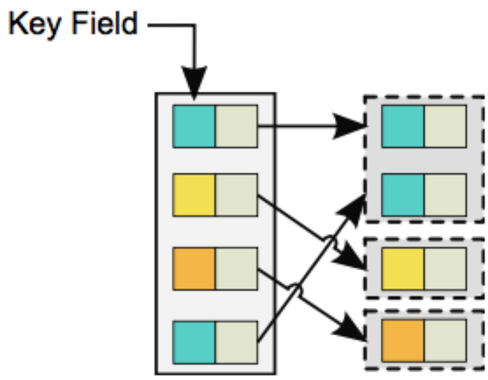
Map / FlatMap Operators

- *Record-at-a-Time* operator with one input
  - calls its user-defined function for each individual input record
- Same semantics as Hadoop MapReduce's Map:
  - The user function accepts a single record as input and can emit any number of records (*0* to *n* for FlatMap, 1 for Map).
- Typical applications: filters or transformations



Reduce Operator

- *Group-at-a-Time* operator with one input
  - groups the tuples of its input on a so-called *Record Key* and hands each group into the user function
    - User function accepts list of tuples as input and can emit any number of tuples.
  - Record Key: one or more fields of the input records that implement the *Key* interface.
- Common application: aggregations

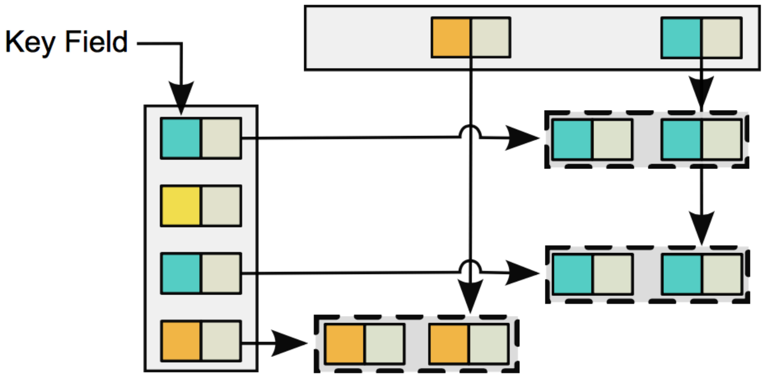


Reduce Optimisation: Combine

- Reduce operator supports <u>optional</u> user-defined *Combine* function
  - Optimisation for the case that the result of the Reduce operator can be computed from partial results that were computed on subgroups
  - E.g. Sum:
    The sum of a list of numbers, say (1 + 2 + 3 + 4), can be computed by adding the sums of subgroups, e.g., (1 + 2) and (3 + 4).
- Beneficial because it reduces the amount of data
  - Combine() is called before data is transferred over the network to establish the full group for the final Reduce call
- Note: Flink does not guarantee that the Combine function is actually executed.
  - This is a cost-based decision done by Flink's optimizer.

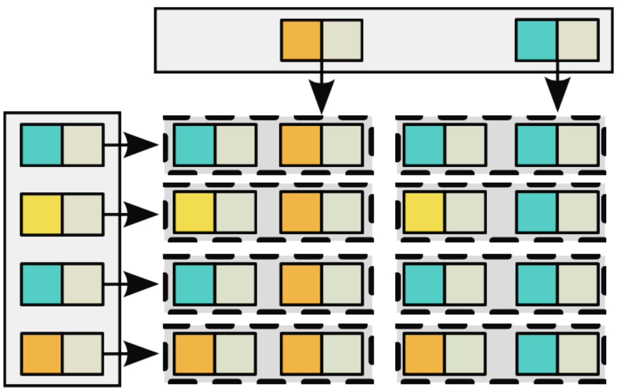
Join Operator

- *Record-at-a-Time* operator with two inputs
  - requires the specification of record keys on both inputs
  - operator function (equi-)joins both inputs on their record keys and hands matching pairs of records into the user function
  - The user function accepts one record of each input and can emit any number of records
- Typical application: equi join



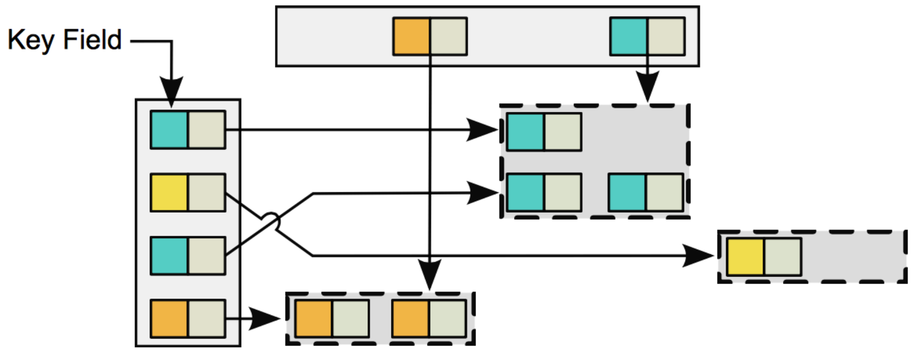
Cross Operator

- *Record-at-a-Time* operator with two inputs.
  - In contrast to Join, Cross does not require the specification of a record key on any input.
  - The operator function builds the Cartesian product of the records of both inputs and calls the user function for each pair of records.
  - The user function accepts one record of each input and can emit any number of records.
- Caution – very expensive operator that makes most sense on small inputs.



CoGroup Operator

- *Group-at-a-Time* operator with two inputs.
  - requires the specification of a record key for each input.
  - Records of both inputs are grouped on their key. Groups with matching keys are handed together to the user function.
    - groups that have no matching group in the other input are given as a list to the user function and the other input list remains empty.
  - The user function accepts one list of records for each input and can emit any number of records.
- Application: Outer Join



Flink Optimizer

- Flink has a dedicated optimizer
- Goal: efficient execution plans for data processing tasks
  - Estimates output sizes of each operator
  - Based on cardinalities, decides on parallelism and local (in-memory) vs. shipping (via-network) pipelining strategy
- Big challenge: UDFs
  - Optimizer it is not aware of the internal semantics of an UDF
  - Programming model allows to provide **annotations** for UDFs that explicitly provide the required information:
    - **Constant fields**: Lists the fields of a record that are NOT modified by the UDF (neither value or position in the record).
    - **All fields constant except**: Lists all fields that are modified by the UDF. All other fields are considered constant.
  - Still, cardinalities and selectivities can only be guessed…

Data Flow

- The Flink programming model is based on data flow of parallelizable operators. The data flow is defined in the form of an *operator DAG* that consists of:
  - **Data Sources**
  - **Data Transformations**
  - **Data Sinks**

Data Sources

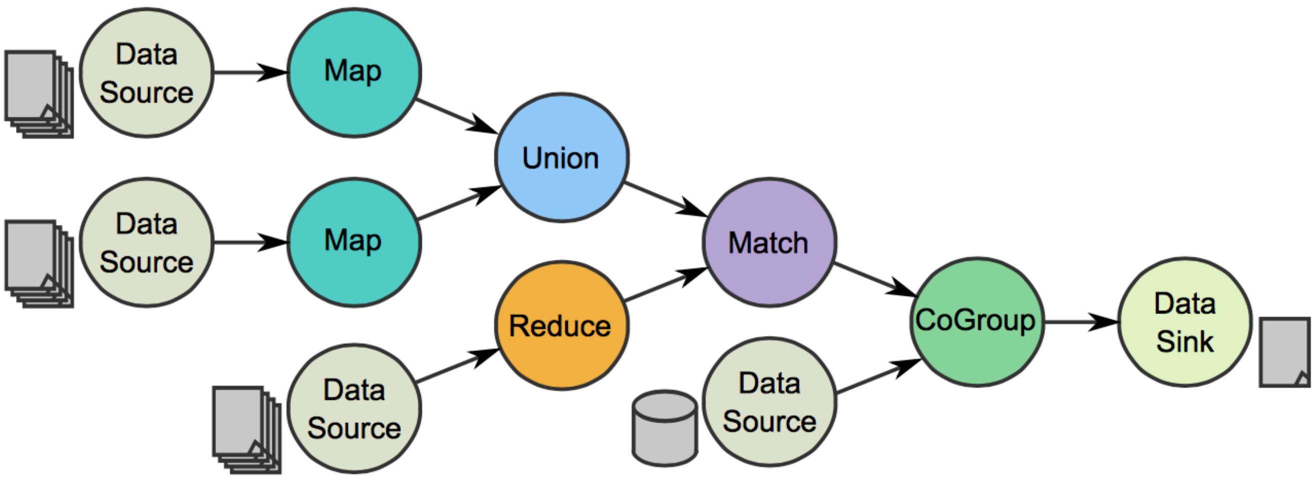
- A *data source* is the entry point of data into a data flow
  - Can connect to a variety of data stores, e.g. HDFS or relational or NoSQL database systems
  - Produces exactly one **DataSet**
- DS provide a generic interface called *InputFormat*
  - In general, InputFormats are also user-defined functions.
- Predefined InputFormats for a variety of sources including:
  - CSV files,
  - Row-delimited text files,
  - Binary files with constant record length,
  - Collection-based (creates Source by iterating over a collection)
  - Generic input DS eg. on external database sources (such as JDBC)

Data Sinks

- Data sinks are the exit point where data leaves a data flow.
- Similar to data sources, data sinks provide a generic interface called *OutputFormat* to write their input data set to a variety of data stores or streams.
  - An OutputFormat serializes records into a format, such as a textual or binary representation, and writes it to an interface outside of the system as for example a file system or database. OutputFormats are as well user-defined functions such that data can be written to any external data store.
- Flink provides OutputFormats for:
  - CSV files,
  - Row-delimited text files, and
  - Binary files.
  - Print to console

Data Flow Composition

- A data flow is composed of any number of data sources, transformations, and data sinks
  - cyclic data flows are disallowed
- Iterative data flows possible why **Iterate** operator



Example: Java API

- Cf. http://ci.apache.org/projects/flink/flink-docs-release-0.8/programming_guide.html
- http://ci.apache.org/projects/flink/flink-docs-release-0.8/examples.html
- https://github.com/apache/flink/tree/master/flink-examples/flink-java-examples/src/main/java/org/apache/flink/examples/java

```
public class WordCount {

   public static void main (String[] args) {
      // set up the execution environment
      final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

      DataSet<..> source = …

      // Operations on the data set go here
      DataSet<..> plan = source. …

      // define where to emit result of the execution plan
      plan.writeAsCSV(outPath, "\n", " "); // alternative on-srceen oputput:  plan.print();

      // initiate the actual execution
      env.execute("WordCount Example");
   }
}
```

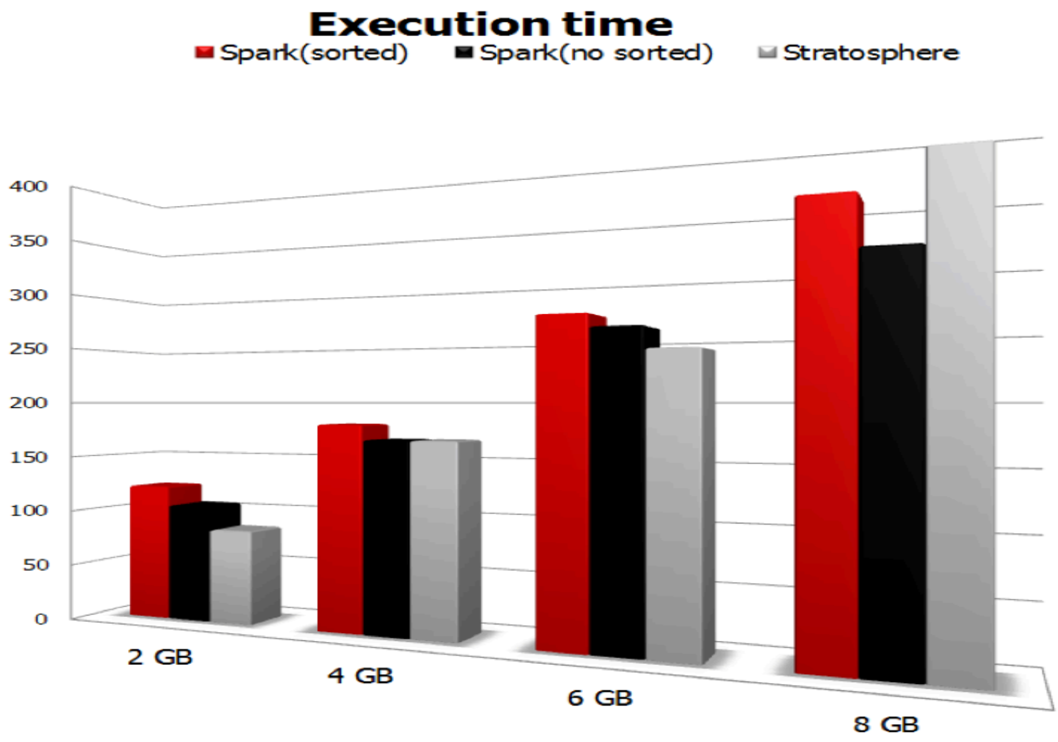

Example: Word Count

```
public class WordCount {

  public static void main(String[] args) throws Exception {

    // set up the execution environment
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // get input data
    DataSet<String> text = env.fromElements(
        "To be, or not to be,--that is the question:--",
        "Whether 'tis nobler in the mind to suffer",
        "The slings and arrows of outrageous fortune",
        "Or to take arms against a sea of troubles,"
        );

    DataSet<Tuple2<String, Integer>> counts =
        // split up the lines in pairs (2-tuples) containing: (word,1)
        text.flatMap(new LineSplitter())
        // group by the tuple field "0" and sum up tuple field "1"
        .groupBy(0)
        .aggregate(Aggregations.SUM, 1);

    // emit result
    counts.print();

    // execute program
    env.execute("WordCount Example");
  }
}
```

```
public class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
    // normalize and split the line into words
    String[] tokens = value.toLowerCase().split("\\W+");

    // emit the pairs
    for (String token : tokens) {
      if (token.length() > 0) {
        out.collect(new Tuple2<String, Integer>(token, 1));
      }
    }
  }
}
```

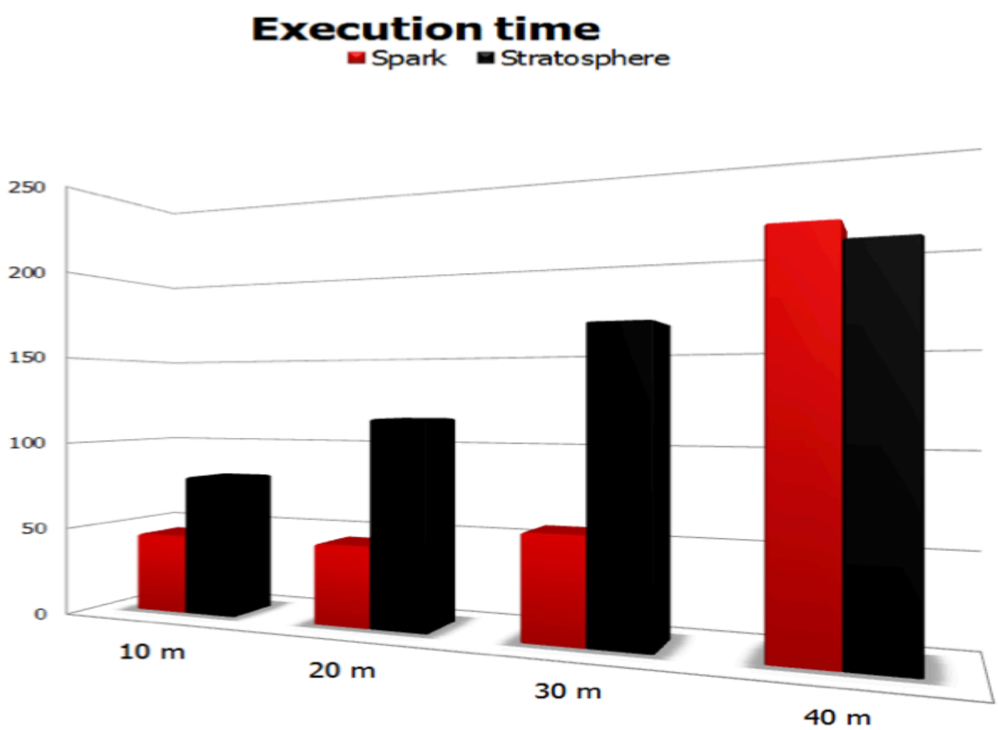
http://ci.apache.org/projects/flink/flink-docs-release-0.8/java_api_quickstart.html
http://ci.apache.org/projects/flink/flink-docs-release-0.8/examples.html

Performance Comparison (WordCount)



Word Count task on various data sizes in HDFS. [Ze Ni, 2013]

Performance Comparison (K-Means)



K-means algorithm with 2 iterations on various data sizes (millions of points) in HDFS. [Ze Ni, 2013]

Conclusions

- Several approaches exist for more sophisticated data analytics on top of Hadoop/HDFS infrastructure
  - strive to make data processing easier to program (more high-level)
  - support for iterative and streaming data processing
  - while keeping scalability and fault tolerance of MR
- Apache Spark:
  - Core idea: make distributed datasets a first-class primitive to provide a simple, efficient programming model for stateful data analytics
- Apache Flink:
  - Powerful data flow language on top of MapReduce
  - Integrates many ideas from traditional data processing, more focus on automated plan optimisation

References

- Spark / BDAS:
  - Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica:"Spark: cluster computing with working sets". In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud 2010.
  - M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica: "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing". In *USENIX* NSDI, 2012.
  - Ion Stoica: "Berkeley Data Analytics Stack (BDAS) Overview", Strata Conf., Feb 2013.
  - Matei Zaharia, Mosharaf Chowdhury, Justin Ma, Michael Franklin, Scott Shenker, Ion Stoica: "Spark: In-Memory Cluster Computing for Iterative and Interactive Applications", amplab.
  - Mike Franklin: "The Berkeley Data Analytics Stack: Present and Future", NICTA, Feb 2014.
- Flink / Stratosphere:
  - http://flink.apache.org
  - http://stratosphere.eu/docs/
  - Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke: "Massively parallel data analysis with PACTs on Nephele". *PVLDB* 3:1625–1628, Sept 2010.
- Ze Ni: "Comparative Evaluation of Spark and Stratosphere", master thesis, KTH Stockholm, 2013.

Lecture Outlook

- Today:
  - Working of Assignment 1 in the lab rooms in SIT with tutor feedback
  - Submission tomorrow (Fri) by 6pm in Blackboard
  - Self-Reflection Survey to be filled in too
- Next Week: Details on Spark and FLink
  - Including lab
  - A2 to be published mid of Week 8/ early Week 9 too
- Friday this week (28nd April): Data Centre Excursion
  - Visit to one of the Equinix data centres in Alexandria (200 Bourke Rd)
  - 2 tours @ 1-1:30pm and @ 1:45pm-2:30pm
    => ENROL IN TOUR GROUPS IN Blackboard

Usability Study of Cloud Computing Frameworks for Big Data Analytics

This semester, we also conduct a study of the usability and 'learnability' of cloud computing frameworks (such as Hadoop, Spark, Flink).

You can opt-in to use your assignment submissions and the self-reflection survey data for our study.

The study outcome will **not** be used for marking and everyone will need to do the same tasks; but if you opt-in, the **anonymised meta-data** of your assignment submissions and the answers from the self-reflective survey will be used for the study.

More details will be send by email later today.