15-319 / 15-619 Cloud Computing

Recitation 14

December 1st 2015

Overview

- **Recent Tasks reflection**
  - Project 4.1
  - Quiz 11
- **Budget issues**
  - Tagging, 15619Project
- **This week's schedule**
  - Unit 5 - Modules 20 & 21
  - Project 4.2
  - 15619Project Phase 3
- **Demo**
- **Twitter Analytics: The 15619Project**

Reminders

- Monitor AWS expenses regularly and tag all resources
  - Check your bill (Cost Explorer > filter by tags).
- Piazza Guidelines
  - Please tag your questions appropriately
  - Search for an existing answer first
- Provide clean, modular and well documented code
  - Large penalties for not doing so.
  - **Double check** that your code is submitted!! (verify by downloading it from TPZ from the submissions page)
- Utilize Office Hours
  - We are here to help (but not to give solutions)
- Use the team AWS account and tag the 15619Project resources carefully

Project 4.1 FAQ

- End-to-End Application using MapReduce, H-Base and web frontend
  - Text Corpus -> NGrams -> Language Model
  - Web app querying HBase
  - Extending ideas for Character-grams
- FAQs
  - Unable to load data into HBase from Reducer, MapReduce program hangs randomly.
  - Ans: Use the correct jars, build on the instance with the right dependencies, try on small datasets first
- Secret to MapReduce: Start small

Module to Read

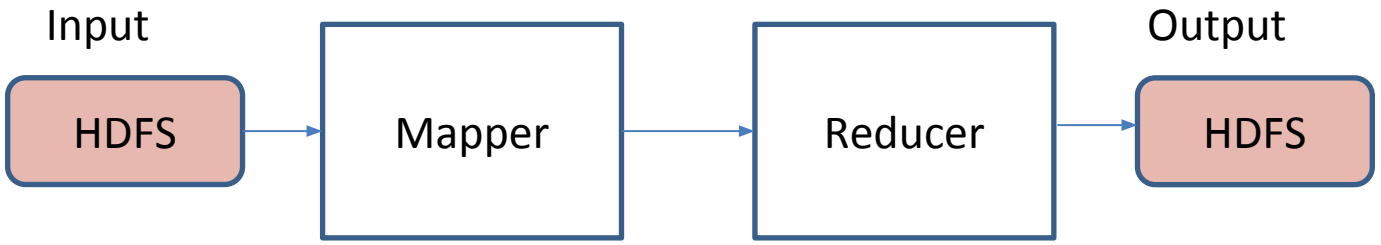
- UNIT 5: Distributed Programming and Analytics Engines for the Cloud
  - Module 18: Introduction to Distributed Programming for the Cloud
  - Module 19: Distributed Analytics Engines for the Cloud: MapReduce
  - Module 20: Distributed Analytics Engines for the Cloud: Spark
  - Module 21: Distributed Analytics Engines for the Cloud: GraphLab

Project 4

- Project 4.1
  - MapReduce Programming Using YARN
- Project 4.2
  - Iterative Programming Using Apache Spark 
- Project 4.3
  - Stream Processing using Kafka/Samza

Typical MapReduce Job

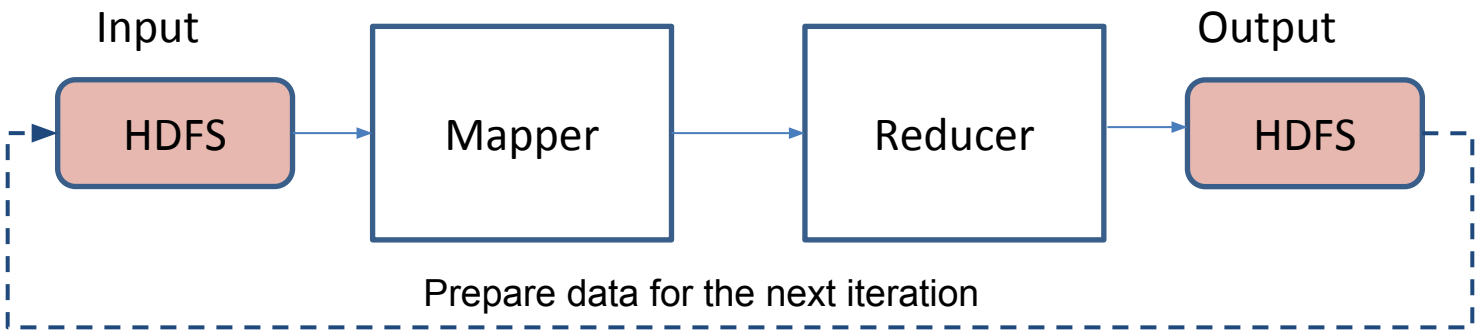
- Simplistic view of a MapReduce job



- You simply write code for the
  - Mapper
  - Reducer
- Inputs are read from disk and outputs are written to disk
  - Intermediate data is spilled to local disk

Iterative MapReduce Jobs

- Some applications require iterative processing
- Eg: Machine Learning, etc.



- MapReduce: Data is always **spilled** to disk
  - Added overhead for each iteration
  - Can we keep data in memory? Across Iterations?
  - How do you manage this?

Resilient Distributed Datasets (RDDs)

- RDDs are
  - can be in-memory or on disk
  - read-only objects
  - partitioned across the cluster
    - partitioned across machines based on a range or the hash of a key in each record

Operations on RDDs

- Loading

```
>>>input_RDD = sc.textFile("text.file")
```

- Transformation
  - Apply an operation and derive a new RDD

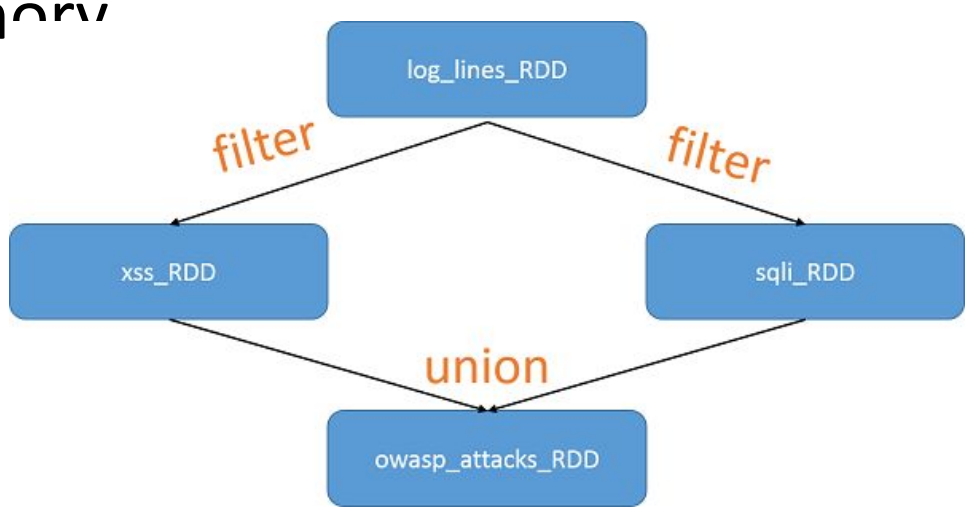
```
>>>transform_RDD =  input_RDD.filter(lambda x: "abcd" in x)
```

- Action
  - Computations on an RDD that return a single object

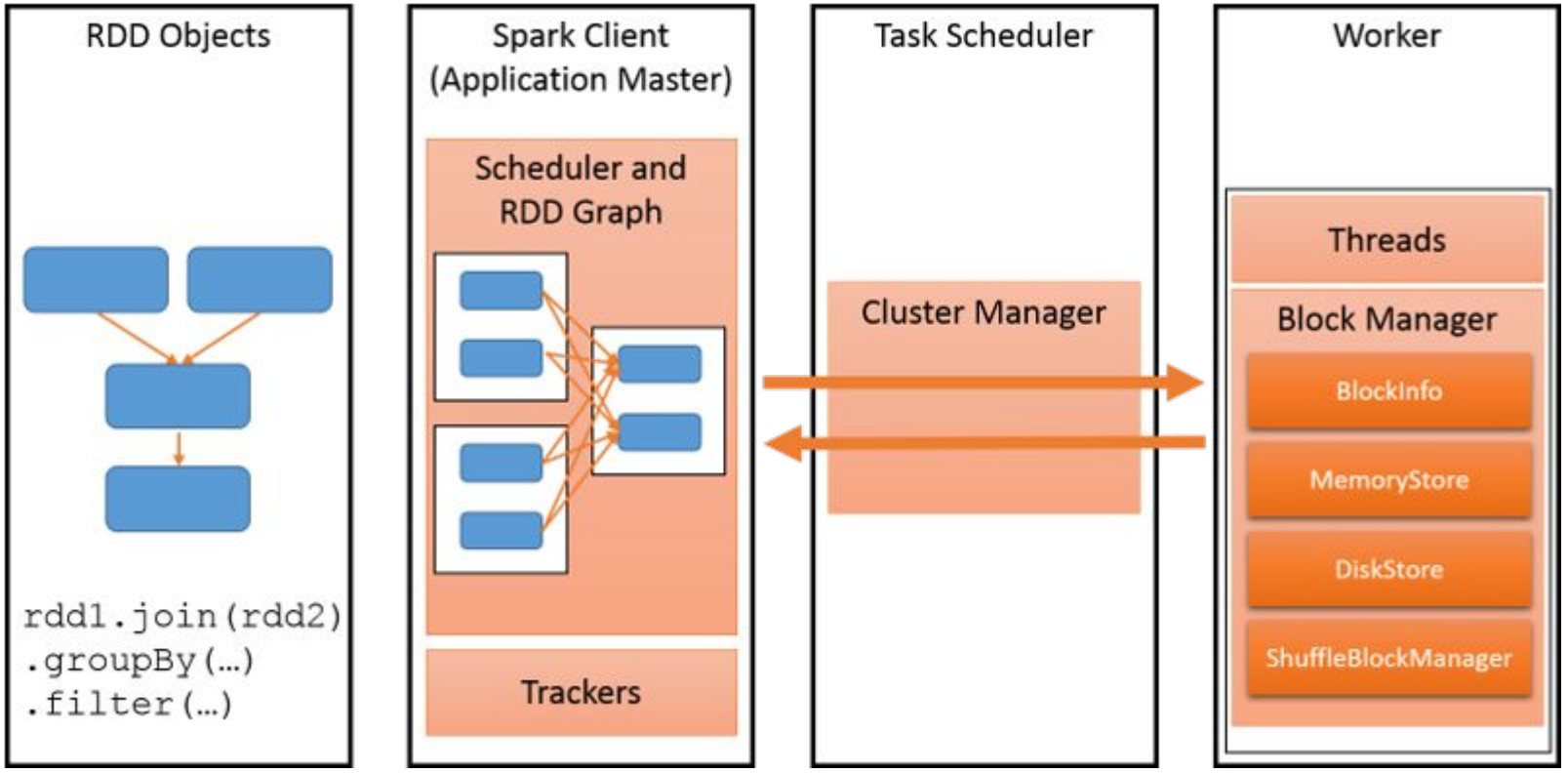
```
>>>print "Number of “abcd”:" + transform_RDD.count()
```

RDDs and Fault Tolerance

- Actions create new RDDs
- Instead of replication, recreate RDDs on failure
- Use RDD lineage
  - RDDs store the transformations required to bring them to current state
  - Provides a form of resilience even though they can be in-memory



The Spark Framework

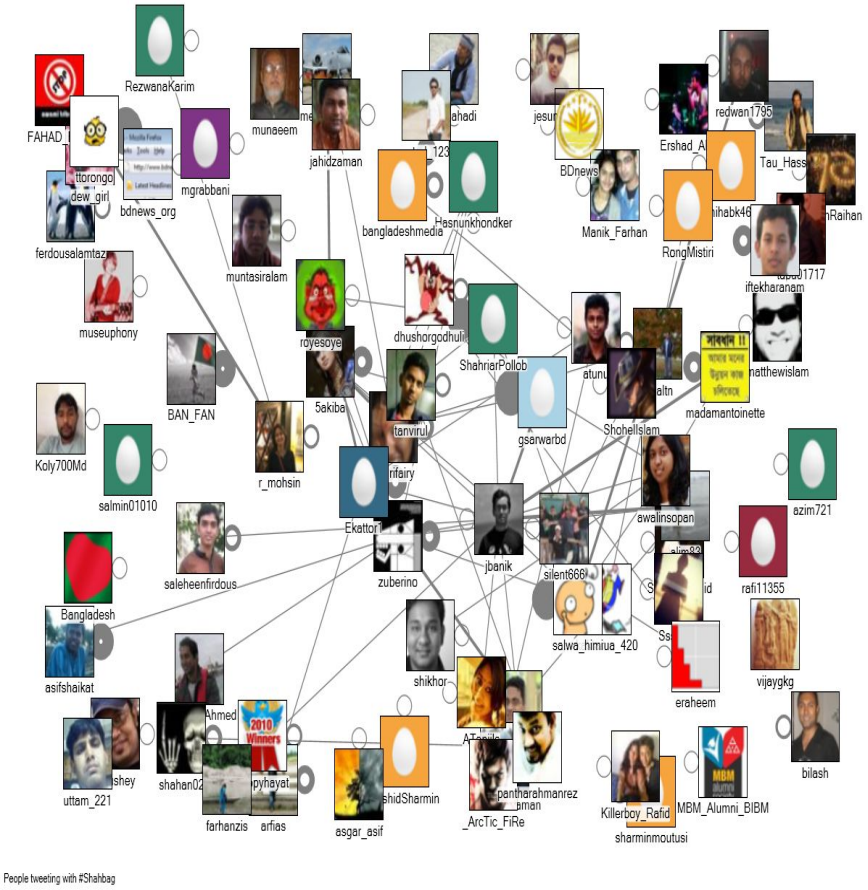


Spark Ecosystem

- Spark SQL
  - Allows running of SQL-like queries against RDDs
- Spark Streaming
  - Run spark jobs against streaming data
- MLlib
  - Machine learning library
- GraphX
  - Graph-parallel framework

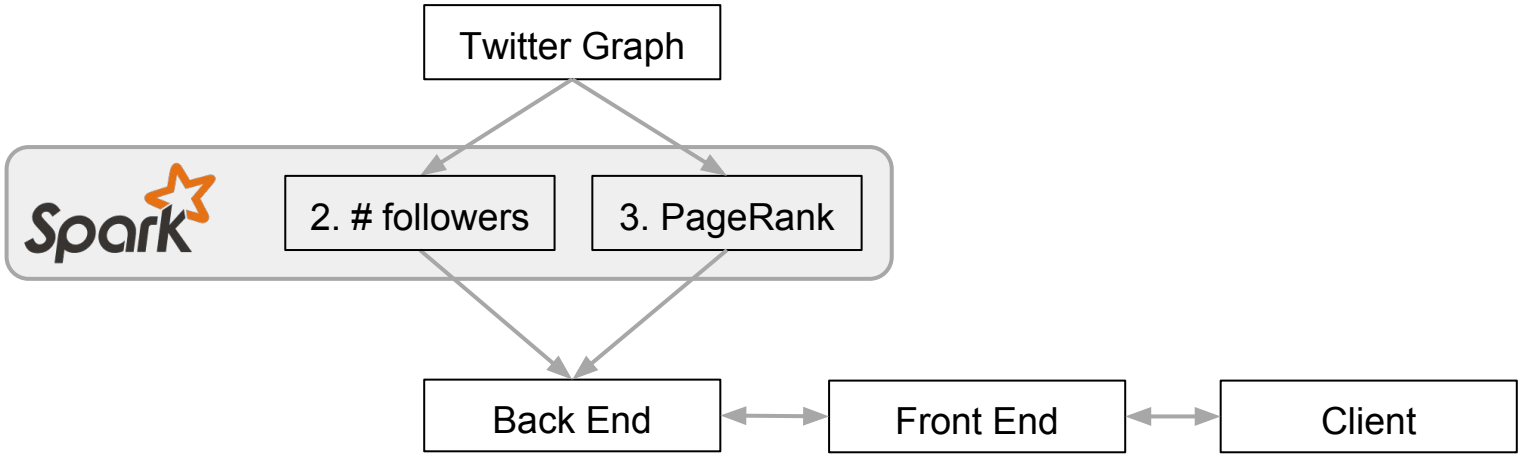
Project 4.2

- Use Spark to analyze the Twitter social graph
  - Number of nodes and edges
  - Number of followers for each user
  - Run PageRank to compute the influence of users



Project 4.2 - Overview

- Use the Twitter social graph dataset
- Analyze the social graph with Spark
- Find the influence of users and rank them with PageRank



Project 4.2 - Three Parts

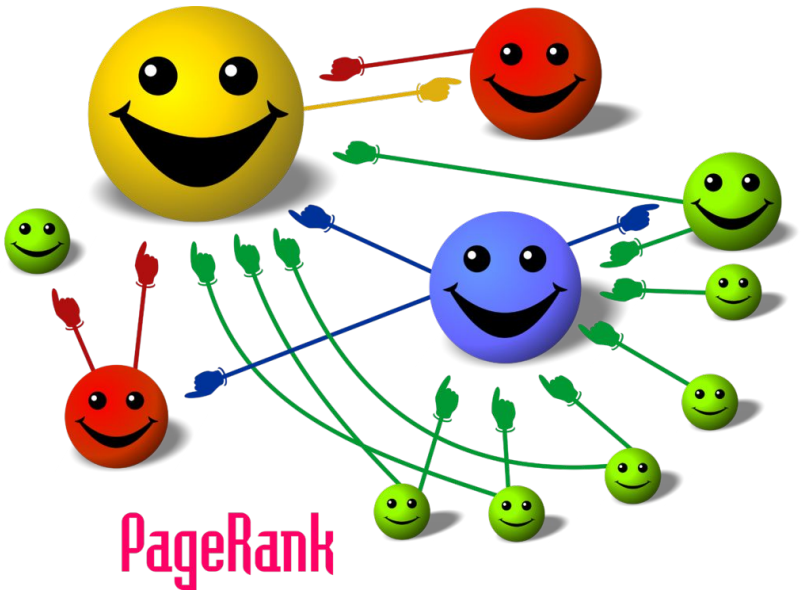
1. Enumerate the Twitter Social Graph
   - Find the number of nodes and edges
   - Edges in the graph are directed. (u, v) and (v, u) should be counted as two edges

2. Find the number of followers for each user

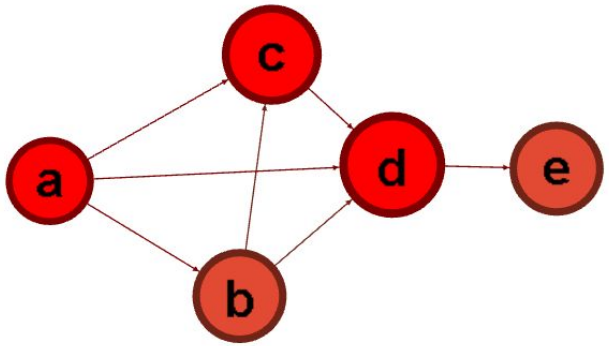
3. Rank each user by influence
   - Run PageRank with 10 iterations
   - Need to deal with dangling nodes

PageRank

- Give pages ranks (scores) based on links to them
- A page that has:
  - Links from many pages ⇒ high rank
  - Link from a high-ranking page ⇒ high rank



PageRank



- For each Page i in dataset, Rank of i can be computed:

$$\text{Rank}[V_x] = (1-d) + d\left(\sum_{i=1}^{n} \frac{\text{Rank}[V_i]}{C[V_i]}\right)$$

where V_x is Vertex x, d is a damping factor,
and V_i is one of the n neighboring vertices of V_x,
and $C[V_i]$ is the count of the neighbors of Vertex V_i

- Iterate for 10 iterations
- Formula to be implemented for 4.2 is slightly more complex. Read carefully!!!

PageRank in Spark (Scala)

(Note: This is a simpler version of PageRank, than P4.2)

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS)
{
    // Build an RDD of (targetURL, float) pairs
    // with the contributions sent by each page
    val contribs = links.join(ranks).flatMap
    {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }

    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) => x+y)
                    .mapValues(sum => a/N + (1-a)*sum)
}
```

Launching a Spark Cluster

- Use the Spark-EC2 scripts
- Command line options to specify instance types and spot pricing
- Spark is an in-memory system
  - test with a single instance first
- Develop and test your scripts on a portion of the dataset before launching a cluster

Spark Shell

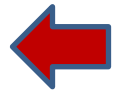
- Like the python shell
- Run commands interactively
- Demo in second half of recitation
- On the master, execute (from /root)
  - ./spark/bin/spark-shell
  - ./spark/bin/pyspark

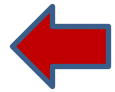
Grading

- Submit your work in the submitter instance
- Don’t forget to submit your code
- For Task 1
  - Put your answers in the answer file
  - Run submitter to upload your answer
- For Task 2
  - Load your result into the **follower** table in database
  - Run webserver and use submitter to submit
- For Task 3
  - Load your result into the **pagerank** table in database
  - Run webserver and use submitter to submit

Upcoming Deadlines

- Quiz 12 : Unit 5 - Modules 20 & 21
  - Open: 12/04/2015 12:01 AM Pittsburgh
  - Due: 12/04/2015 11:59 PM Pittsburgh
- Project 4.2 : Iterative Programming with Spark
  - Due: 12/06/2015 11:59 PM Pittsburgh
- 15619Project : Phase 3
  - Live-test due: 12/02/2015 4:59 PM Pittsburgh
  - Code and report due: 12/03/2015 11:59 PM Pittsburgh





Busy Week Coming Up!

Wednesday	Thursday	Friday	Sunday
Wednesday 12/2/2015 18:00:01 EST • Phase 3 Live Test	Thursday 12/3/2015 23:59:59 EST • Phase 3 Code & Report Due	Friday 12/4/2015 23:59:59 EST • Quiz 12	Sunday 12/6/2015 23:59:59 EST • P4.2 Due

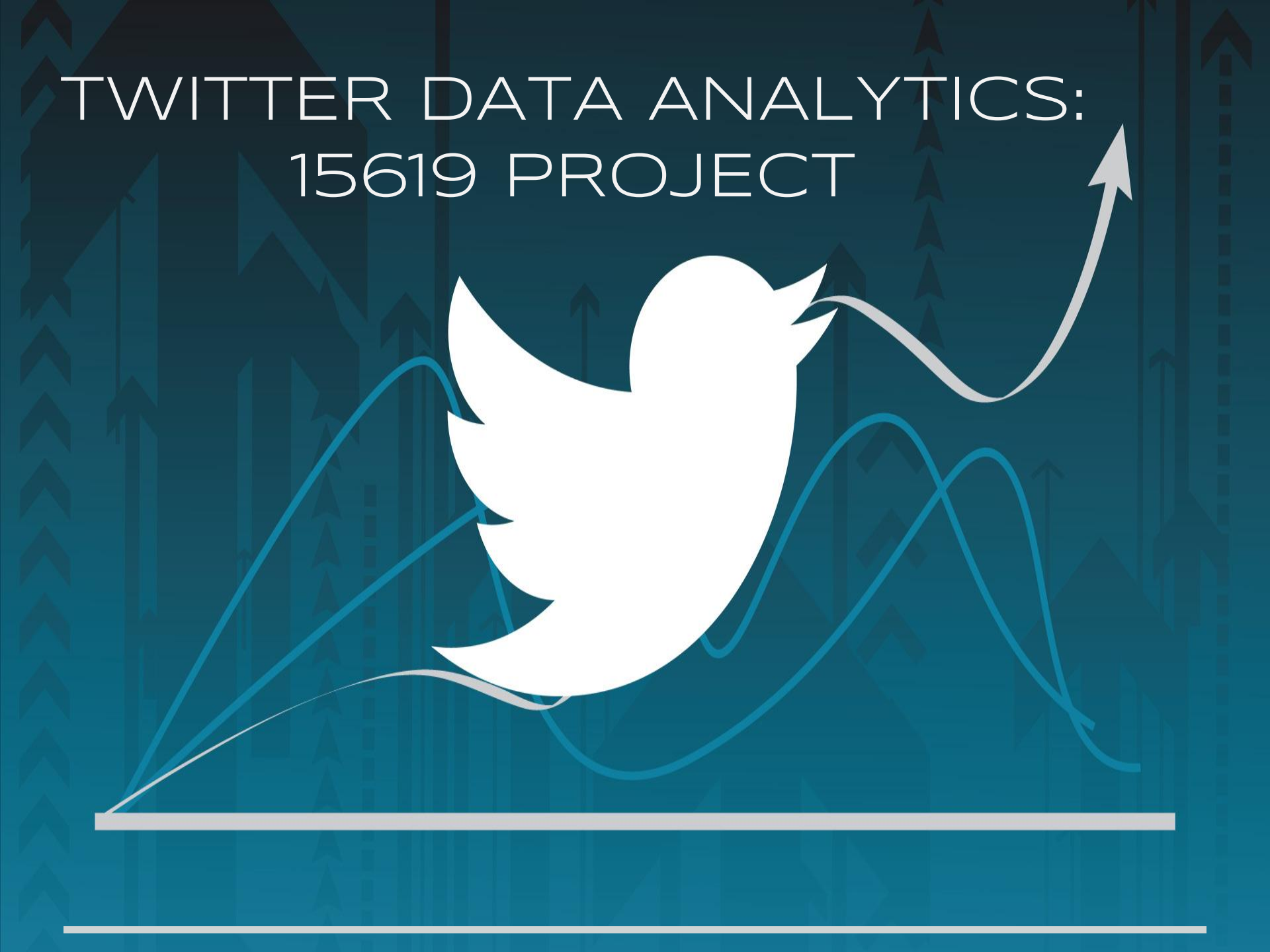
Don’t forget the deadlines!

Project 4.2

- Demo

Questions?

TWITTER DATA ANALYTICS: 15619 PROJECT



15619Project Agenda

- Query 5 Discussion
- Query 6 Discussion
- Upcoming Deadlines
- Phase 3 Live Test

Query 5: Tweet Counter

- **Description:** The query asks for the total number of tweets sent by all users given a range of userids.
- **Request:** We send you two user ids

```
GET /q5?userid_min=u_id&userid_max=u_id
```

- **Response:** Your web service needs to return the number of tweets sent within the range of user ids where user ids are inclusive
- **Warning:** Ignore duplicate tweet IDs (Count once)

Query 5: Tweet Counter

```
GET /q5?userid_min=2&userid_max=10
```

User ID	Tweet ID
1	101
1	102
2	103
2	104
3	105
3	105
4	106
7	107
10	108

Response Format:

```
TEAMID,TEAM_AWS_ACCOUNT_IDs\n
Count\n
```

Guess the Response:

```
TEAMID,TEAM_AWS_ACCOUNT_IDs\n
6\n
```

Query 5: Suggestions & Clarifications

- No filtering based on time (as in Q2)
- Remove duplicate tweets
- Q5 input user id's are inclusive
- Ignore malformed user id
- Explore techniques to flatten data (Reduce query latency)

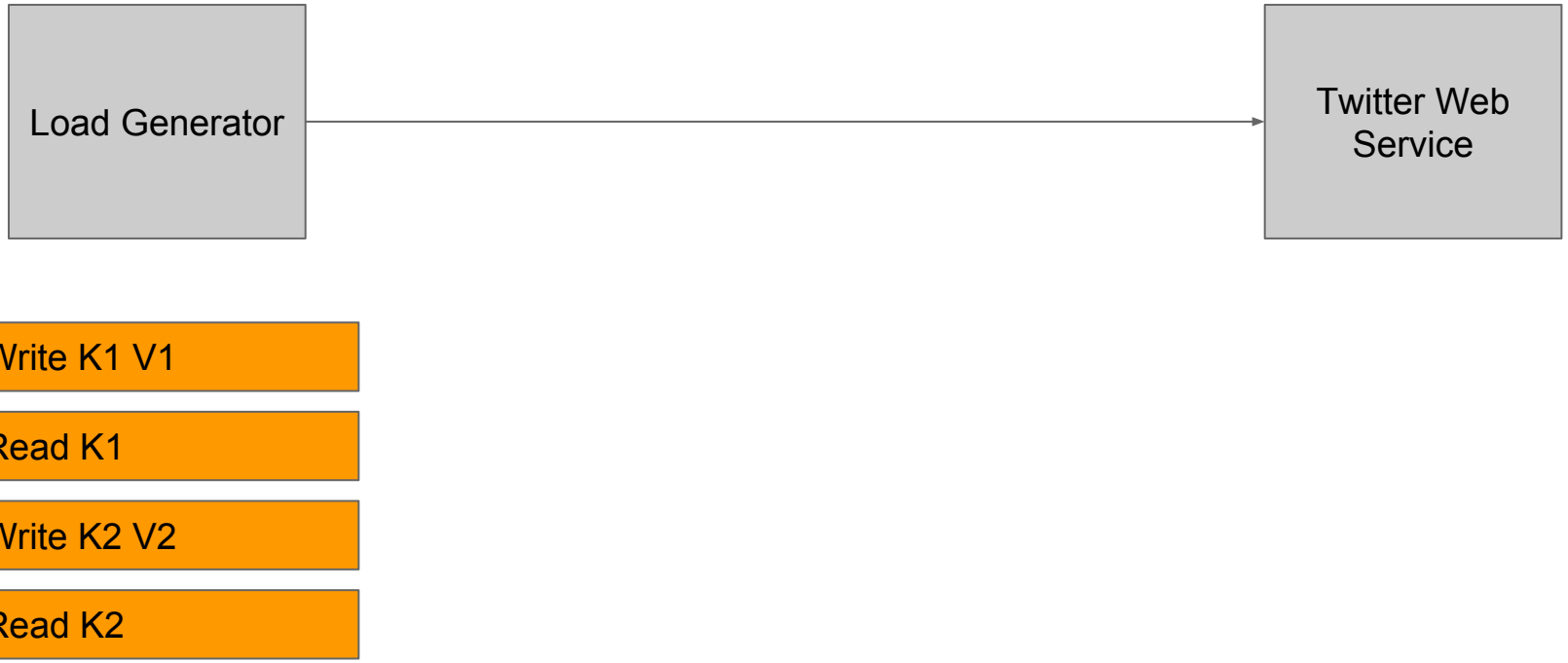
Query 6: Tweet Tagger

Finally, we're dealing with writes!!!

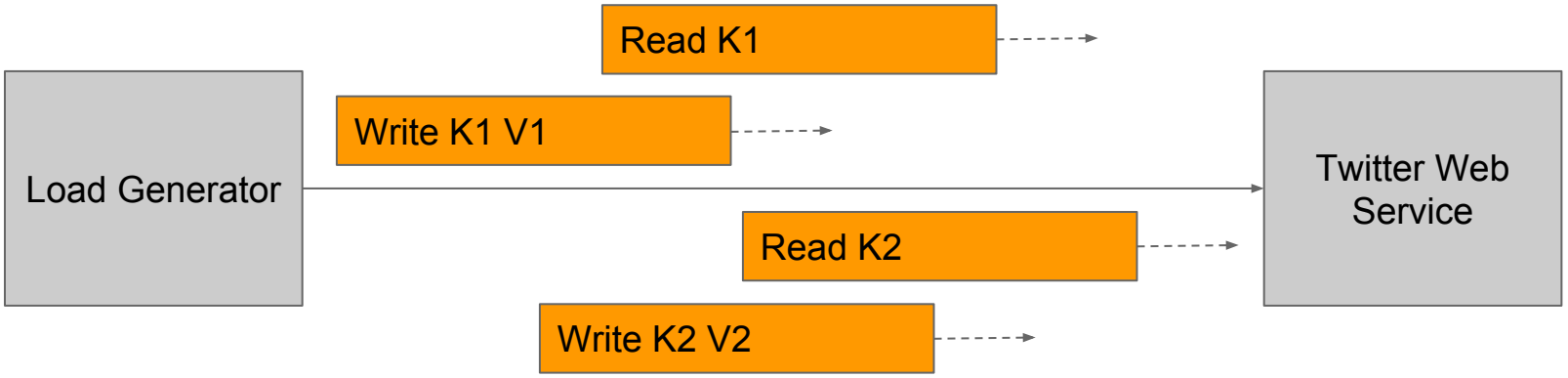
- Append a random string to the end of an existing tweet
- **Each tweet can have only a single appended tag at a time** (last writer wins)
- ETL similar to /q2 (with no date limits)
- When we read in Q6, we expect to see the censored tweet text, with an uncensored appended tag (if any)
- Correctness test strictest for reads

Query 6: Tweet Tagger

- Problem: Request Reordering

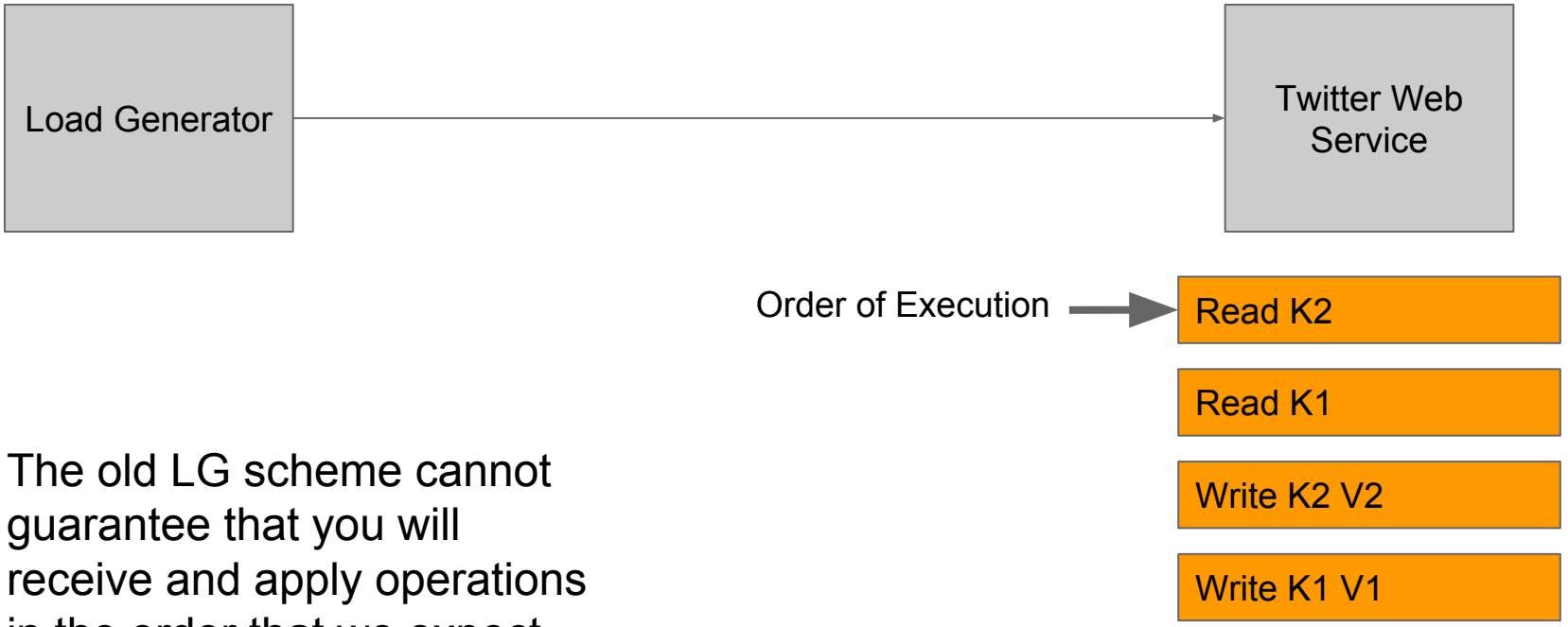


Query 6: Tweet Tagger



Network delay causes a read to happen before a write, which was not expected by the grader

Query 6: Tweet Tagger

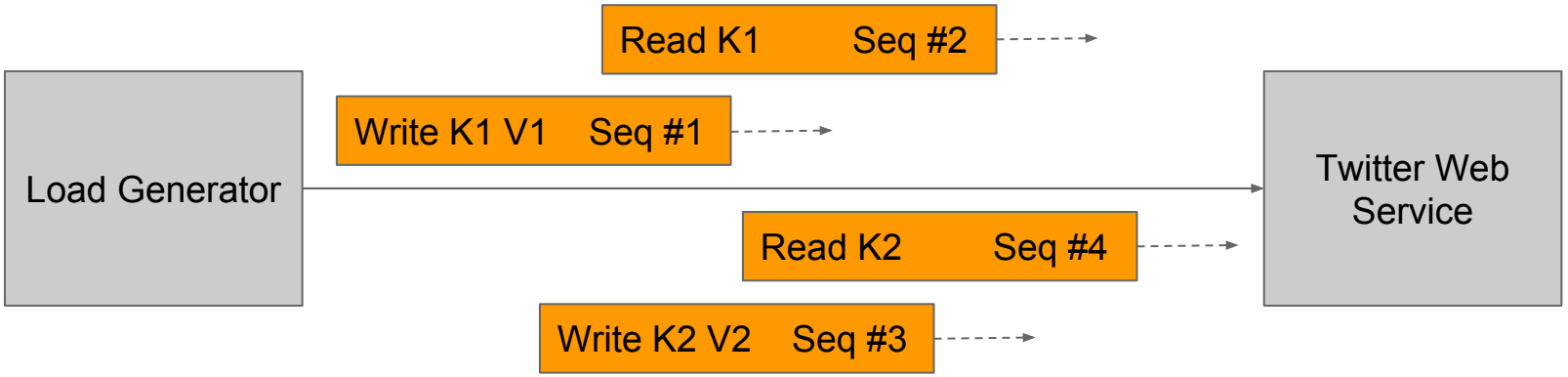


- The old LG scheme cannot guarantee that you will receive and apply operations in the order that we expect

We have a problem with the request reordering

Query 6: Tweet Tagger

Solution: Sequence Numbers



Write K1 V1 Seq #1

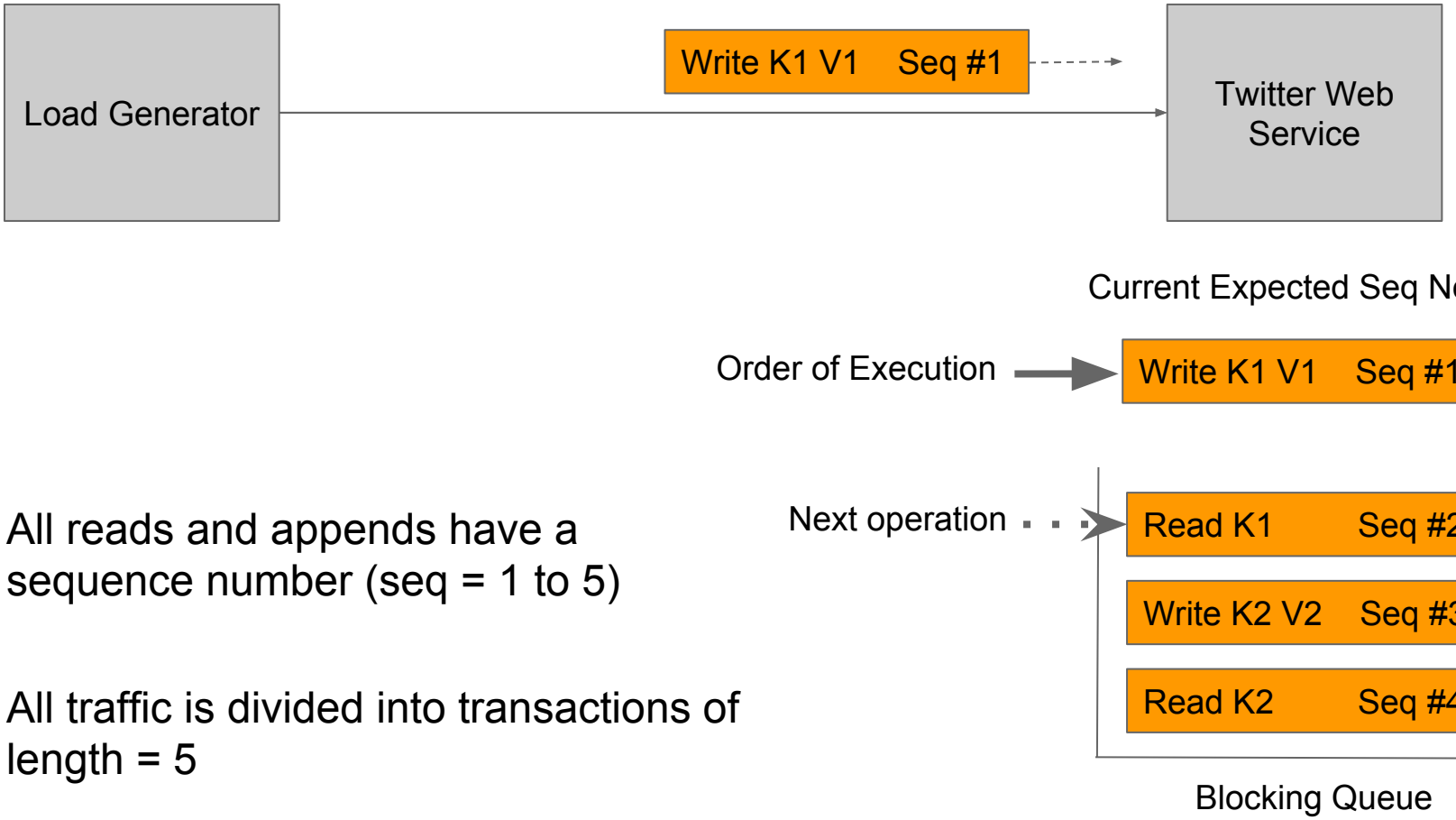
Read K1 Seq #2

Write K2 V2 Seq #3

Read K2 Seq #4

Query 6: Tweet Tagger

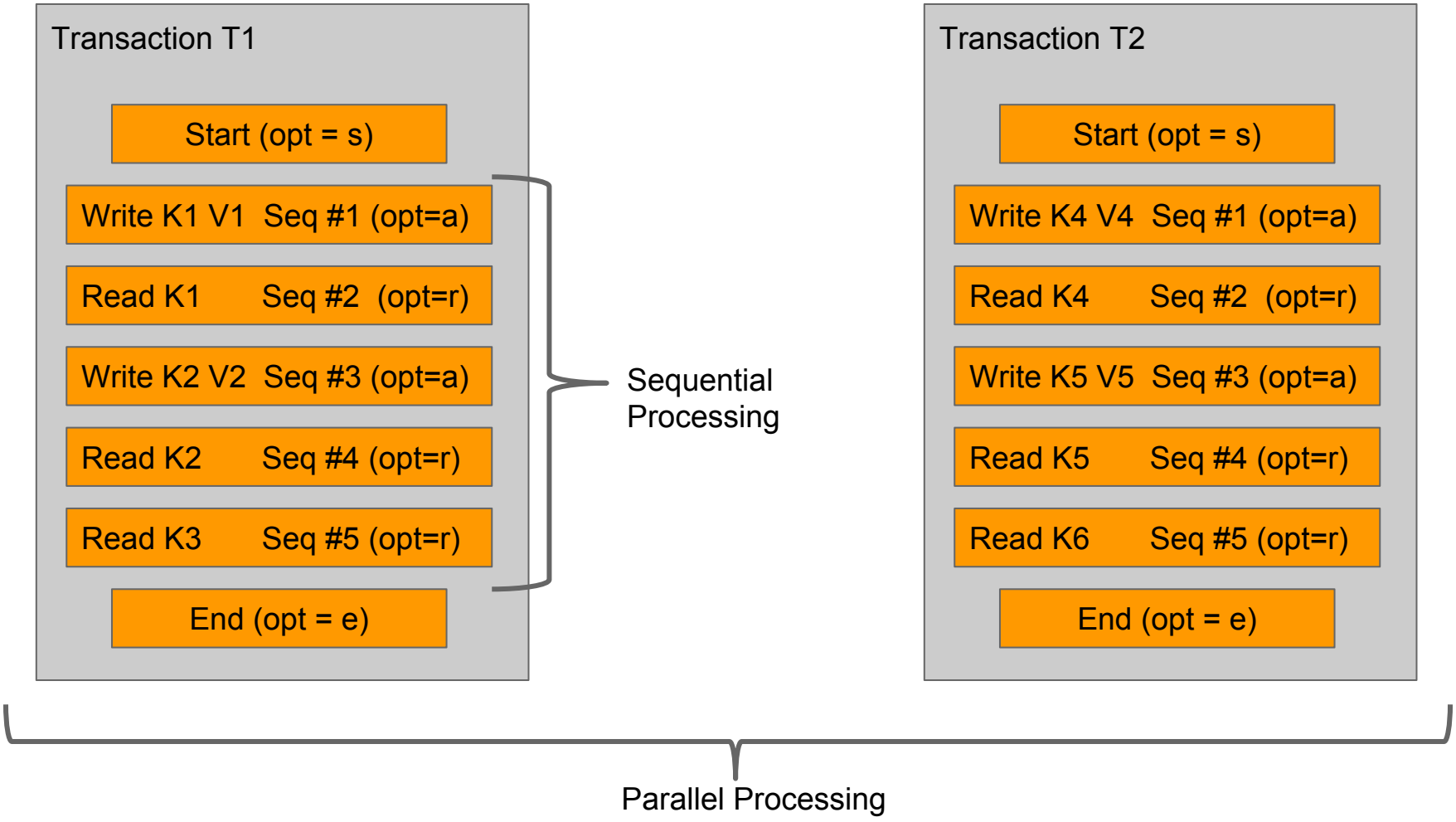
Solution: Sequence Numbers



- All reads and appends have a sequence number (seq = 1 to 5)
- All traffic is divided into transactions of length = 5

Query 6: Tweet Tagger

Transaction



Query 6: Tweet Tagger

- Designing a replicated backend
  - Ideally, ensure that a write updates all replicas before reading from any replica
  - Faster: Only read from the most “recently updated replica”
  - Or: Update all replicas asynchronously (for ideas, see chain replication, other schemes in Ceph)
  - Tradeoffs: Accuracy v/s Performance

Query 6: Tweet Tagger

- Designing a sharded backend
  - Split data between nodes based on keys
  - Benefit: More space/memory efficient

- ELB
  - If you are using ELB:
    - Your front-end may need to be node-aware
    - Extra hop?

  - If not using ELB:
    - Consider nginx or HAProxy or other LBs

Query 6: Tweet Tagger

Consider Tweet ID: **44898831041785037O**

@Maria_LeonPL chulada de mujeres....sensacional paisana... estaremos atento de su intervención... besos tu caballero de la noche¡¡

Query 6: Tweet Tagger

- Step 1 : Start transaction (**opt=s**)

```
/q6?opt=s&tid=3000001

TEAMID,TEAM_AWS_ACCOUNT_ID\n

0\n
```

- Hint:
  - All transactions operate on an independent set of tweet IDs

Query 6: Tweet Tagger

- Step 2 : Exactly 5 Appends (**opt=a**) or Reads (**opt=r**)

```
/q6?
tid=3000001&seq=1&opt=a&tweetid=4489883104178503
70&tag=ILOVE15619!12
TEAMID,TEAM_AWS_ACCOUNT_ID\n
ILOVE15619!12\n
```

- Hint:
  - When opt=a, return the tag to the user
  - Scope for optimization? Yes, but be careful!!!

Query 6: Tweet Tagger

- Step 2 : Exactly 5 Appends (**opt=a**) or Reads (**opt=r**)

```
/q6?
tid=3000001&seq=2&opt=r&tweetid=448988310417850370
TEAMID,TEAM_AWS_ACCOUNT_ID\n
@Maria_LeonPL chulada de mujeres....sensacional
paisana...estaremos atento de su intervención...
besos tu caballero de la noche¡¡ILOVE15619!12\n
```

Query 6: Tweet Tagger

- Step 2 : Exactly 5 Appends (**opt=a**) or Reads (**opt=r**)

```
/q6?
tid=3000001&seq=4&opt=a&tweetid=448988310417850371&tag=ILOVE15619!13
TEAMID,TEAM_AWS_ACCOUNT_ID\n
ILOVE15619!13\n
```

- Note:
  - If you receive an operation out of order, you need to ensure that the previous operation is performed first
  - Multiple tweet IDs may be operated on in a single transaction

Query 6: Tweet Tagger

- Step 2 : Exactly 5 Appends (**opt=a**) or Reads (**opt=r**)

```
/q6?
tid=3000001&seq=3&opt=a&tweetid=448988310417850371&tag=ILOVE15619!14
TEAMID,TEAM_AWS_ACCOUNT_ID\n
ILOVE15619!14\n
```

- Note:
  - If you receive an operation out of order, you need to ensure that the previous operation is performed first
  - Multiple tweet IDs may be operated on in a single transaction

Query 6: Tweet Tagger

- Step 2 : Exactly 5 Appends (**opt=a**) or Reads (**opt=r**)

```
/q6?
tid=3000001&seq=5&opt=r&tweetid=448988310417850370
TEAMID,TEAM_AWS_ACCOUNT_ID\n
@Maria_LeonPL chulada de mujeres....sensacional paisana...estaremos atento de su intervención... besos tu caballero de la noche¡¡ILOVE15619!12\n
```

Query 6: Tweet Tagger

- Step 3 : End Transaction (**opt=e**) or Reads (**opt=r**)

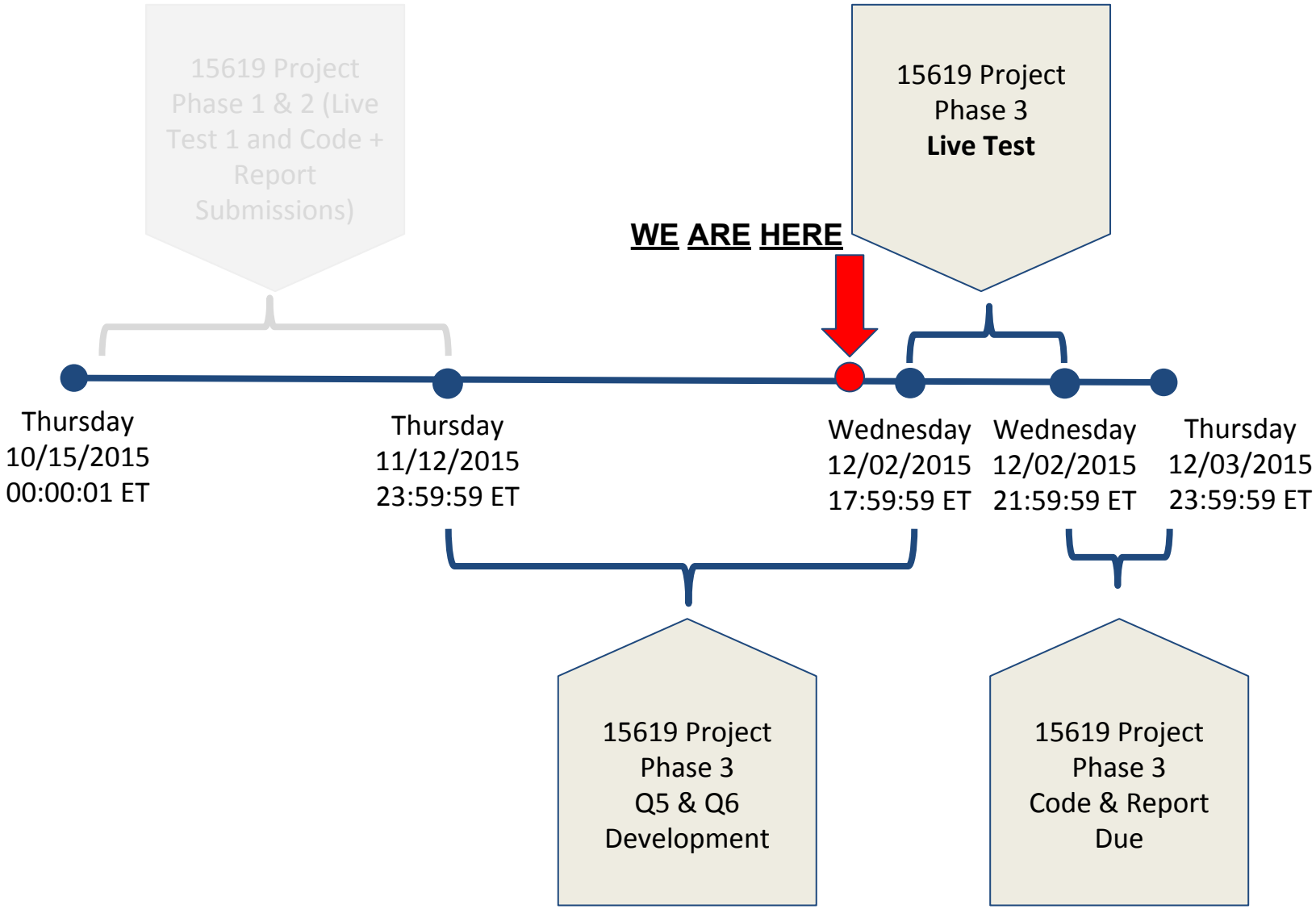
```
/q6?tid=3000001&opt=e

TEAMID,TEAM_AWS_ACCOUNT_ID\n
0\n
```

- Note:
  - Multiple simultaneous, overlapping transactions
  - Ensure that all 5 sequence numbers are handled

Query 6: Suggestions & Clarifications

- Censorship of tweet text before append
- No time filtering
- Tag is not required to be censored
- The appended tag is short (max 14 character)
- Transaction sequence is restricted between 1 to 5
- You can only submit 1 DNS for the live test
- Multiple appends on same tweet
  - Always return latest tag
  - If tag already appended in previous test, replace it with new tag
- Q6 is not in mixed queries

15619 Project Phase 3 Deadlines



What's due next?

- Phase 3 Deadline
  - **Submission of one URL by 16:59 ET (Pittsburgh) Wed 12/2**
    - **Live Test from 6 PM to midnight ET**
  - Choose any one (or both) databases
  - Can only use m1.large or cheaper t1, t2, m1, m3 instances
  - Fix Q1, Q2, Q3, Q4 if your Phase 2 did not go well
  - New queries Q5 and Q6.
  - Phase 3 counts for **60%** of the 15619Project grade

Phase 3 Report [VERY IMPORTANT]

- Start early
- Document your steps
- Identify and isolate the performance impact of each change you make
- Document your ideas and experiments

MAKE A QUANTITATIVE, DATA-DRIVEN REPORT

15619Project Phase 3 Live Test

- 30 minutes warm-up (Q1 only)
- 3 hours Q1 - Q6
- 30 minutes mix-Q1+Q2+Q3+Q4+Q5
- Preparing for the live test
  - Choose a database based on your observations from previous phases and all six queries
  - Caching known requests will not work(unless you are smart)
  - Need to have all Qs running at the same time
  - Avoid bottlenecks in mixed queries