Fault Tolerance in K3

Ben Glickman, Amit Mehta, Josh Wheeler

Outline

- Background
- Motivation
- Detecting Membership Changes with Spread
- Modes of Fault Tolerance in K3
- Demonstration

Outline

- **Background**
- Motivation
- Detecting Membership Changes with Spread
- Modes of Fault Tolerance in K3
- Demonstration

Big Data Systems

- Several sources of Big Data
  - Sciences, Healthcare, Enterprise, and more.
- Need systems that scale to the volume of the data
- Single machine *supercomputers* are expensive
- **“Scale-out”** systems have become popular
  - Cluster of affordable machines
  - Massively parallel with communication over a network
  - e.g., MapReduce (Hadoop), Distributed DBMS

Main-Memory Data Systems

- Disk was the bottleneck of the first generation systems
- Motivated a new class of data systems that compute entirely **in-memory.**
  - Cluster provides a large pool of RAM *(TB scale)*
  - Feasible to store entire datasets *(Spill to disk if necessary)*
  - Improves throughput by orders of magnitude
  - e.g., Spark, Stratosphere, etc.

K3 Background

- Programming framework for building shared-nothing main-memory data systems
  - High level language for systems building
  - Compiled into high-performance native code
- Under development at the Data Management Systems Lab at JHU: http://damsl.cs.jhu.edu
- K3 Github: https://github.com/damsl/k3

K3 Programming Model

- Functional-imperative language
  - Currying, Higher-Order functions
  - Mutable variable, loops
- Asynchronous distributed computation
  - *Triggers* act as event handlers
  - Message passing between triggers defines a dataflow
- Collections library
  - High-level operations (*map, filter, fold, etc.*)
  - Fine-grained updates (*insert, update, etc.)*
  - Nested Collections

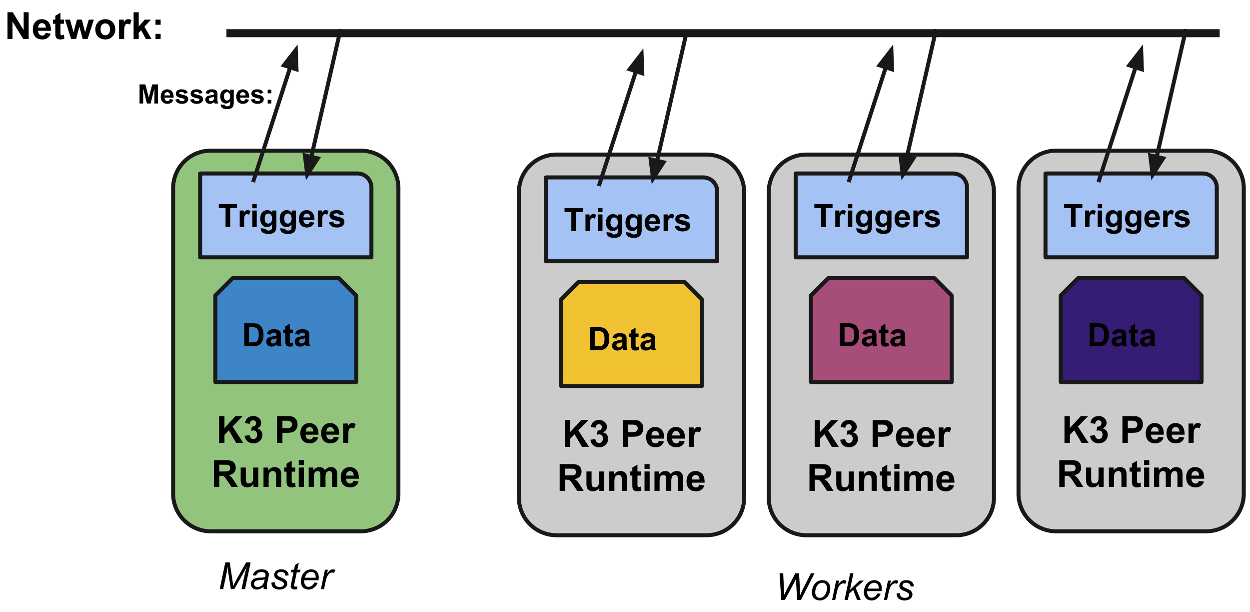
K3 Execution Model

- Several *peers* run the same K3 executable program
- *Shared-Nothing*
  - Each peer can access only its own local data segment
  - Data movement and coordination across peers achieved through *message passing*
- Partitioned Execution Model
  - Large datasets are partitioned and distributed evenly among peers
  - Kept in-memory

K3 Execution Model

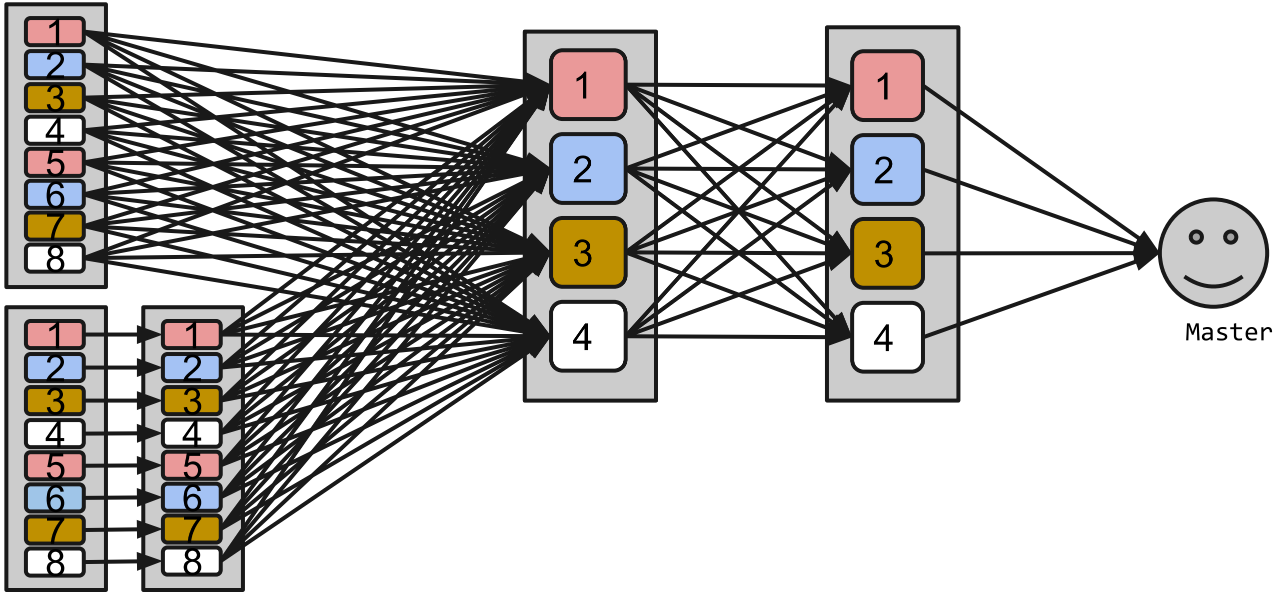
- Focused on large scale analytics *(read-only)* workloads
  - Transactions, fine-grained updates in future work
- Single *master peer* to
  - Coordinate distributed computation
  - Collect results at a single site
- Remaining peers are *workers* that compute over local partitions of a dataset and communicate through messaging

K3 Execution Model



K3 Execution Model

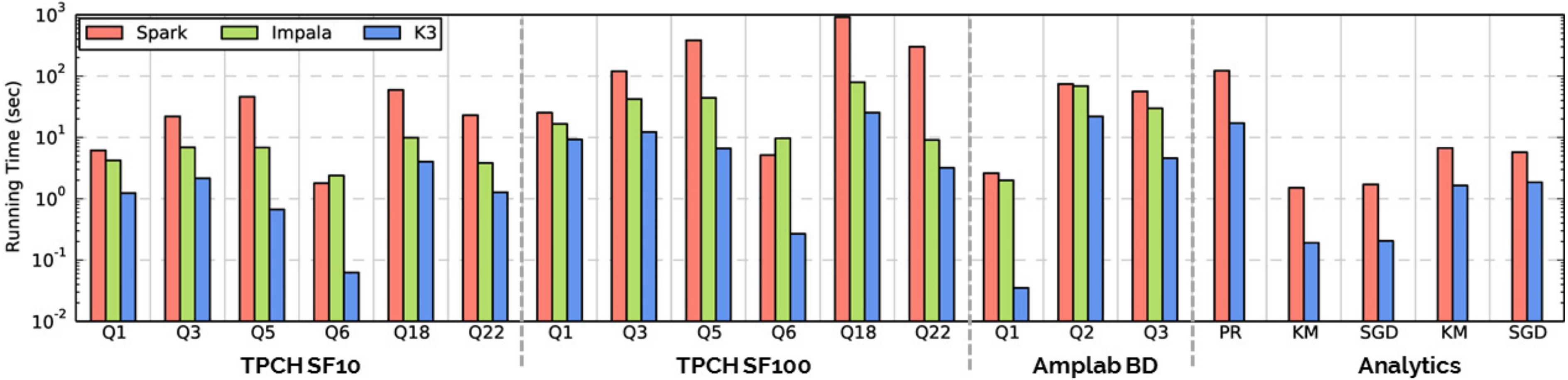
- Used to build multi-stage, complex data-flows:



Data flow for the program that will be demonstrated after the presentation

K3 Performance

- Outperforms two state of the art systems: Spark and Impala
    - SQL processing and iterative Machine Learning and Graph algorithms



K3 Example

Given an *Employees* dataset:

Employee:

name String,
age Integer

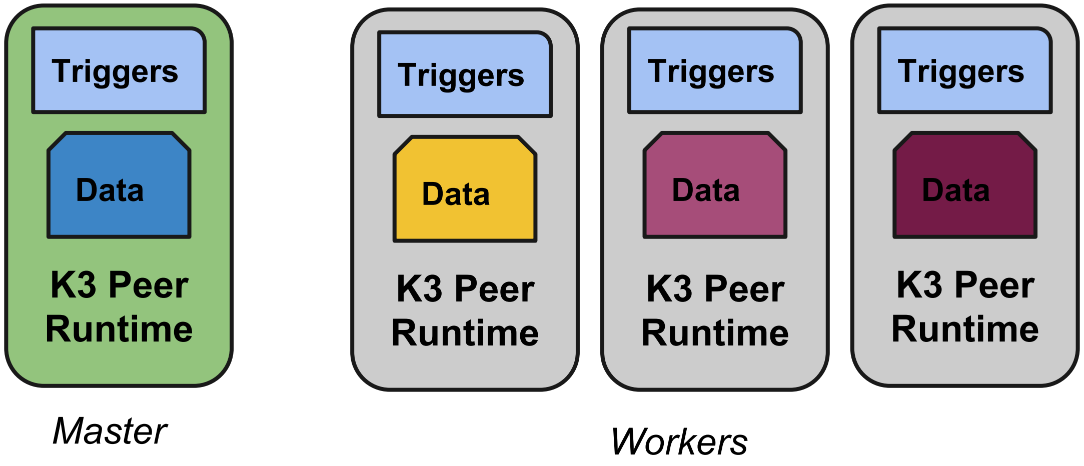
Partitioned among the K3 workers

Bob Smith, 57
...

Saul Goodman, 37
...

Walter White, 67
...

"Find the oldest employee in the dataset"



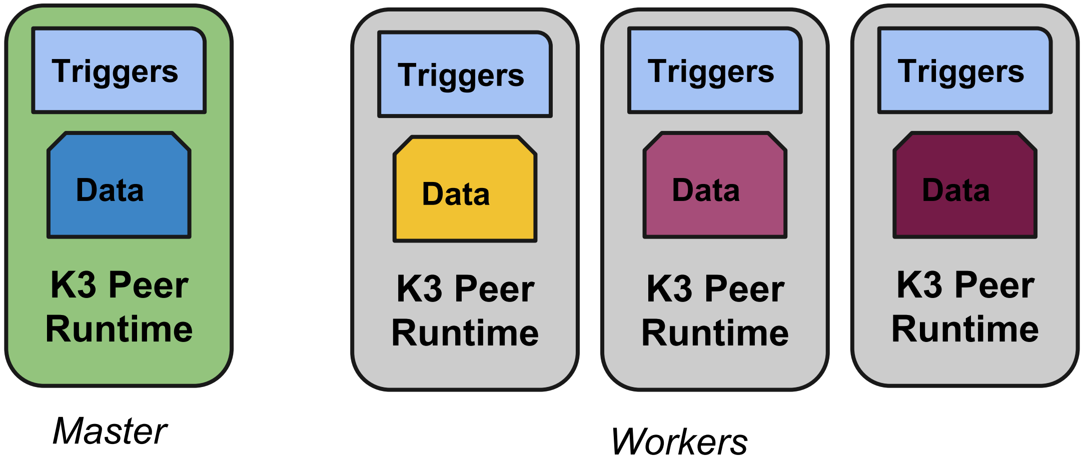
K3 Example

1) **Master: Instruct workers to compute local maximum**

"Find the oldest employee in the dataset"

K3 Code:

```
// Send a message to each peer's
// 'computeLocalMax' trigger
trigger start: () = \_ -> (
  workers.iterate (\p ->
    (computeLocalMax, p.addr) <- ()
  )
)
```



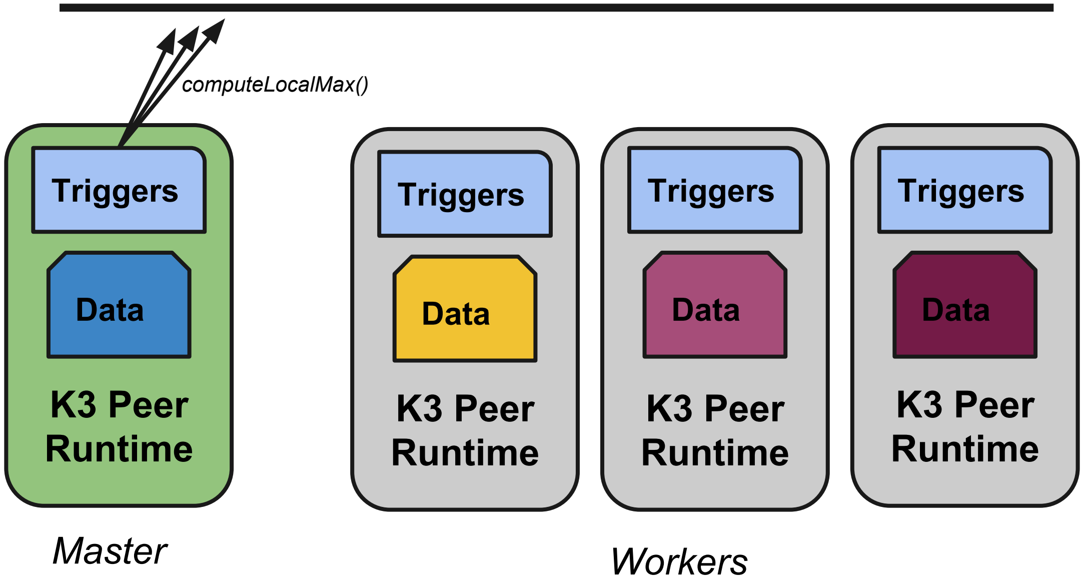
K3 Example

1) **Master: Instruct workers to compute local maximum**

"Find the oldest employee in the dataset"

K3 Code:

```
// Send a message to each peer's
// 'computeLocalMax' trigger
trigger start: () = \_ -> (
  workers.iterate (\p ->
    (computeLocalMax, p.addr) <- ()
  )
)
```



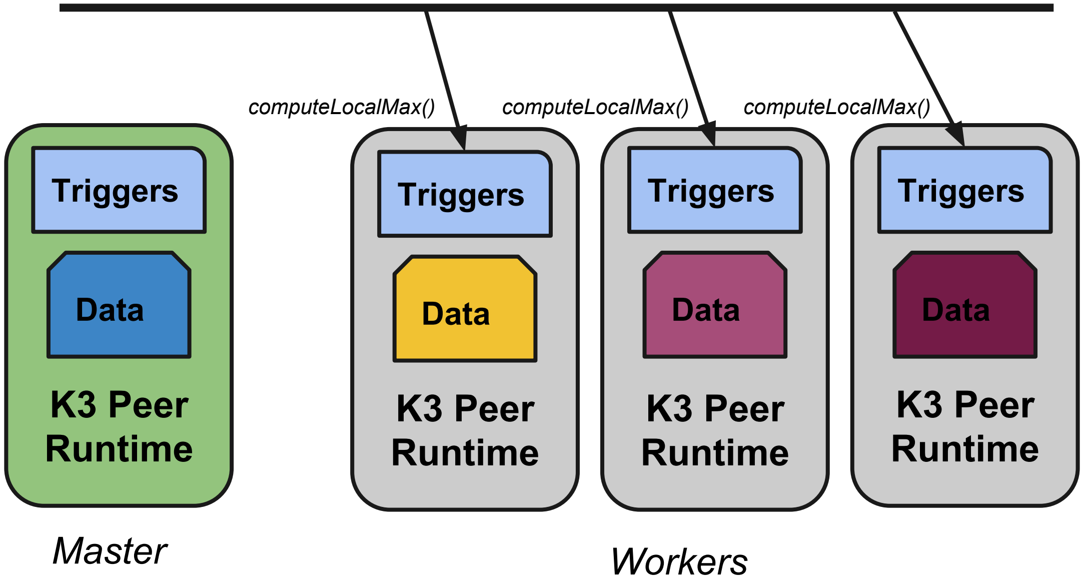
K3 Example

2) Workers: Compute local maximum, send it to the master

K3 Code:

```
// Send local max to the 'collectMax'
// trigger at the master
trigger computeLocalMax: () = \_ ->
  let max = local_data.fold
      (\acc -> \elem ->
         if elem.age > acc.age
           then elem
           else acc
      ) local_data.peek()
  in (collectMax, master) <- max
)
```

"Find the oldest employee in the dataset"



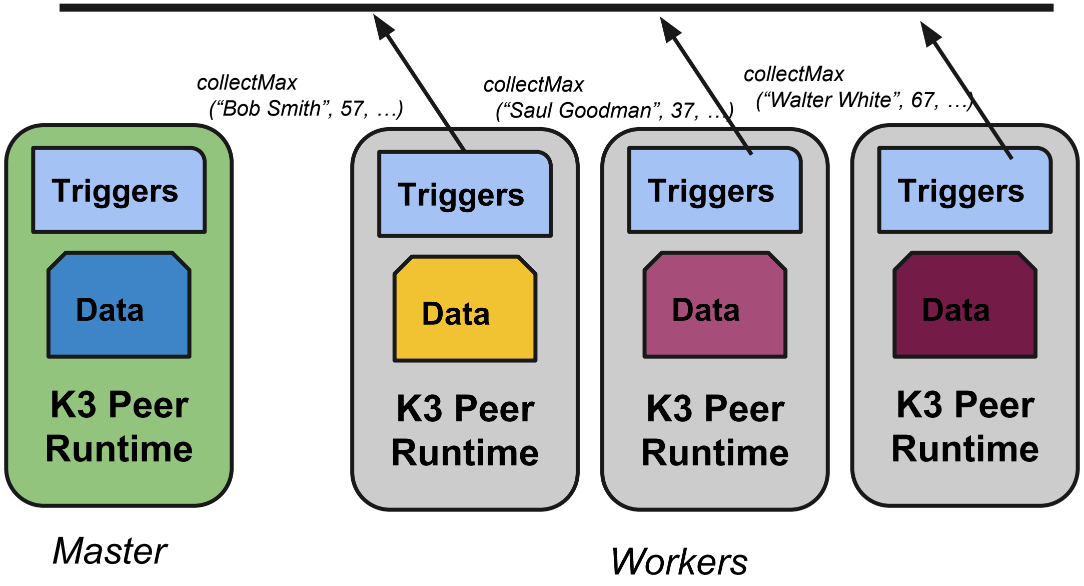
K3 Example

2) Workers: Compute local maximum, send it to the master

K3 Code:

```
// Send local max to the 'collectMax'
// trigger at the master
trigger computeLocalMax: () = \_ ->
  let max = local_data.fold
      (\acc -> \elem ->
          if elem.age > acc.age
            then elem
            else acc
      ) local_data.peek()
  in (collectMax, master) <- max
```

"Find the oldest employee in the dataset"



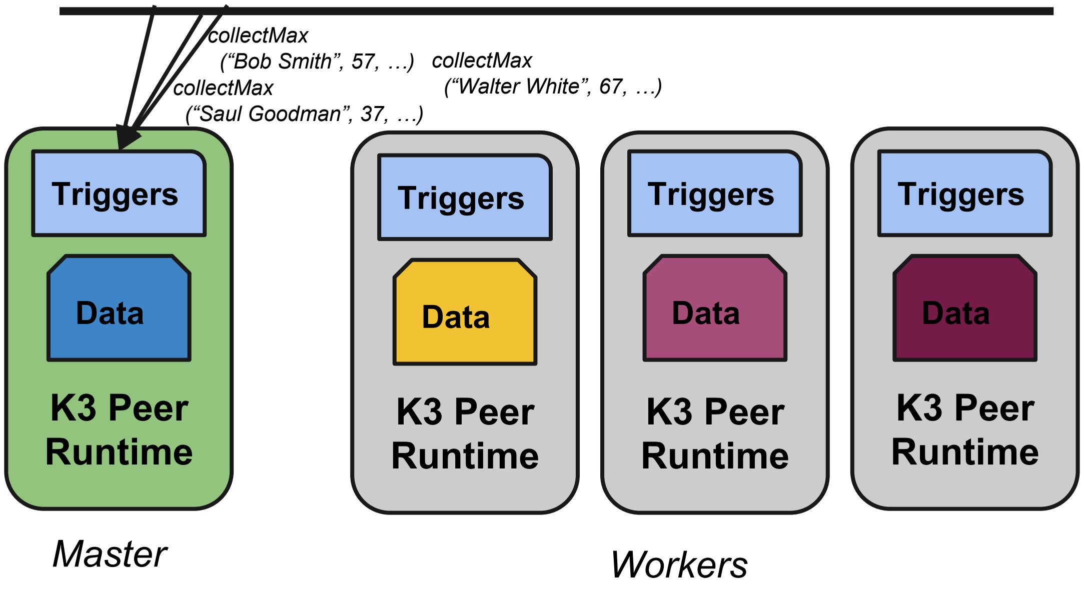
K3 Example

3) Master: Wait to receive all messages, keep track of max.

K3 Code:

```
// Wait to receive from each peer.
trigger collectMax: (string, int) =
  \(name, age) -> (
    global_max =
      if age > global_max.age
        then (name, age)
        else global_max;
    responses_recv += 1;
    if responses_recv==workers.size()
      then print "Finished!"
```

"Find the oldest employee in the dataset"



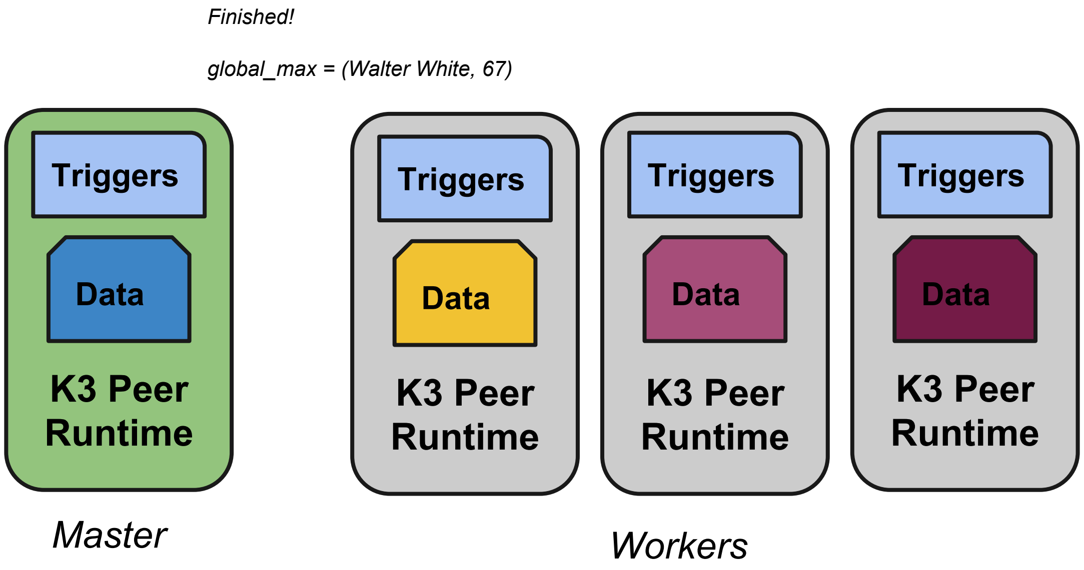
K3 Example

3) Master: Wait to receive all messages, keep track of max.

K3 Code:

```
// Wait to receive from each peer.
trigger collectMax: (string, int) =
  \(name, age) -> (
    global_max =
      if age > global_max.age
        then (name, age)
        else global_max;
    responses_recv += 1;
    if responses_recv==workers.size()
      then print "Finished!"
```

"Find the oldest employee in the dataset"



Outline

- Background
- **Motivation**
- Detecting Membership Changes with Spread
- Modes of Fault Tolerance in K3
- Demonstration

Fault Tolerance Motivation

- Analytical queries are often long-running
  - Large volume of data. Limited CPU throughput
  - Iterative algorithms take time to converge
- Likelihood of failure increases with the number of machines
  - Hardware Failures. Bad Disks, Power Loss, etc.
  - At Largest Scale:
    - Hours of computation
    - Hundreds/Thousands of machines
    - Periodic faults will occur

Mid Query Fault Tolerance

- Without Fault Tolerance: Restart entire computation
  - Any progress made towards a solution before the crash is lost
  - Start from scratch: Hopefully no failures! or else repeat!
- Existing solutions tolerate crashes via
  - Replicated Input Data
  - Optional checkpointing of intermediate program state *(expensive)*
  - Replaying work that has been lost.
    - Hadoop and Spark both replay missing work

Fault Tolerance in K3

- Before our project: K3 did not handle faults
- When a process crashed:
  - Others might become *stuck* waiting for messages
  - Others might attempt to send messages to a missing peer
- Consider the ‘max’ example

K3 Crash Example

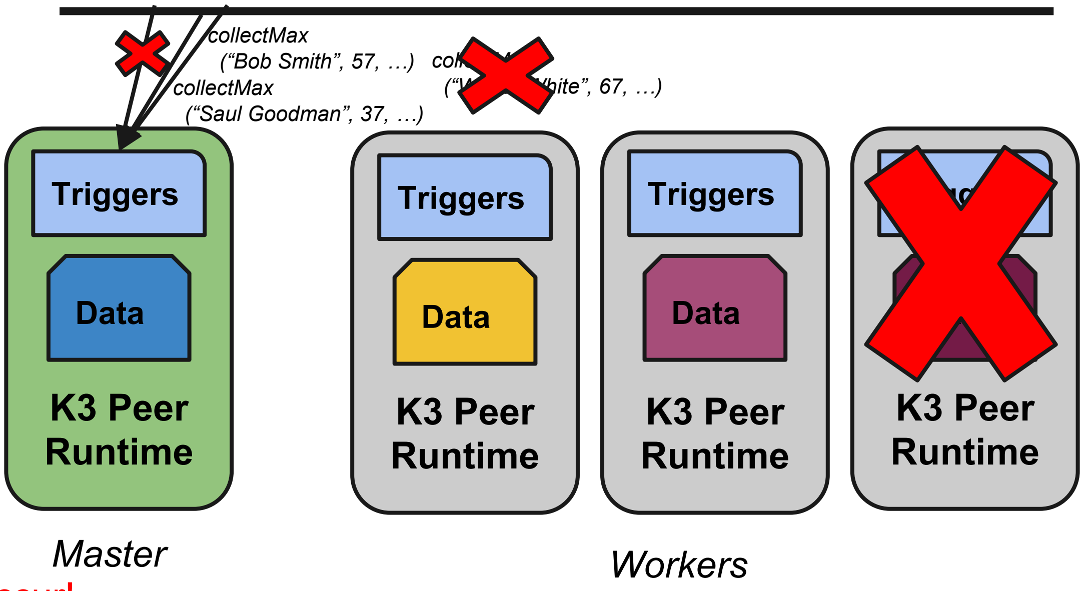
3) Master: Wait to receive all messages, keep track of max.

K3 Code:

```
// Wait to receive from each peer.
trigger collectMax: (string, int) =
  \(name, age) -> (
    global_max =
      if age > global_max.age
        then (name, age)
        else global_max;
    responses_recv += 1;
    if responses_recv==workers.size()
      then print "Finished!"
```

Will never occur!

"Find the oldest employee in the dataset"



Fault Tolerance in K3

- Need to offer programmer a way to react to crashes
- Allow them to implement application specific logic for handling a crash
  - We explored several applications/modes of failure
- Alternatively, a general solution might leverage static analysis of a K3 program to automatically provide fault tolerance
  - Place less burden on the programmer
  - Potential area for future work, not covered in this project

Outline

- Background
- Motivation
- **Detecting Membership Changes with Spread**
- Modes of Fault Tolerance in K3
- Demonstration

Spread

- Group communication toolkit to assist in building reliable distributed systems
- Allows process to join groups for communication
- Processes are alerted when others join or leave groups
  - i.e., due to a process crash or network partition
- We incorporated Spread client into the K3 runtime for its membership functionality

K3 / Spread

- K3 processes act as Spread clients
- K3 command line args specify connection parameters
- Startup protocol:
  - Wait for all processes to join a public group
  - Processes agree on the initial set of *peers* sent to the K3 program
- Spread event loop runs in a separate thread from the K3 event loop
  - Spread client code receives a membership change
  - Creates the appropriate K3 message and injects into program's queue

K3 / Spread

We allow programmers to designate a special *trigger* for handling a membership change

- Indicated with a @:*Membership* Annotation
- Trigger receives set of new members as an argument
- Trigger contains arbitrary application specific logic for reacting to the change
- After startup, called after each membership change

K3 Code:

```
trigger t: [address]@Set = (\members ->
  print “Oh no! A membership change!”;
  ...
) @:Membership
```

Outline

- Background
- Motivation
- Detecting Membership Changes with Spread
- **Modes of Fault Tolerance in K3**
- Demonstration

Fault Tolerance in K3

We explored 3 example models of fault tolerance:

- Terminate gracefully after a crash
- Remaining peers continue after a crash (approximate solution)
- Replay missing work after a crash

Fault Tolerance in K3

We explored 3 example models of fault tolerance:

- **Terminate gracefully after a crash**
- Remaining peers continue after a crash (approximate solution)
- Replay missing work after a crash

Graceful Termination

- Simply exit the program when a membership change is received
- Baby-step towards fault tolerance:
  - All peers are aware of the crash
  - Prevents a peer from becoming ‘stuck’
- General enough for all programs
  - Only prevents ‘stuck’ state
  - Does not help with getting output

K3 Code:

```
trigger t: [address]@Set = (\members ->
  shutdown()
) @:Membership
```

Fault Tolerance in K3

We explored 3 models of fault tolerance:

- Terminate gracefully after a crash
- **Remaining peers continue after a crash (approximate solution)**
- Replay missing work after a crash

Continue After a Crash

For example, make the following changes to prevent the *‘stuck’* state in the ‘max’ program:

- Master keeps track of which peers are expected to respond at any time
  - Instead of counting responses
- After a membership change:
  - Master: Stop expecting messages from any missing peer
  - Workers: Exit if the master has been lost.

Continue After a Crash

- Able to reach an approximate solution
  - Partitions of data have been lost, may affect the answer
  - In the ‘max’ example: the partition containing the true maximum may have been lost.
- Appropriate in certain situations only
  - e.g., training a statistical model
  - Up to the developer to decide if this is acceptable.

Fault Tolerance in K3

We explored 3 models of fault tolerance:

- Terminate gracefully after a crash
- Remaining peers continue after a crash (approximate solution)
- **Replay missing work after a crash**

Recovery by Replay

- Motivated by Spark's *Resilient Distributed Datasets (RDD)*
- Applications that apply coarse-grained transformations to partitioned datasets
    - Many algorithms can be encoded in this model
- Input datasets must be replicated
    - e.g., HDFS replicates input data 3 times

Recovery by Replay

In the RDD model:

- Each partition of data has a *lineage* or set of dependencies
  - Input data comes directly from disk
  - Intermediate data is a result of applying transformations to previously defined partitions
- When a partition is lost, it can be re-computed by replaying its lineage
  - Bottoms-out at disk, if there are still replicas available
  - See example in demonstration

Recovery by Replay

In the RDD model:

- When a machine crashes, the partitions that it was hosting are re-assigned to *several* other machines.
  - Allows the work to be replayed in parallel
- Does not require expensive checkpointing and replication of logs or intermediate datasets
  - A big issue when datasets are large.

Outline

- Background
- Motivation
- Detecting Membership Changes with Spread
- Modes of Fault Tolerance in K3
- **Demonstration**

Proof of Concept

- We Implemented a multi-stage SQL query from the Amplab Big Data Benchmark
  - https://amplab.cs.berkeley.edu/benchmark/ (Query 2)
- Replays missing work in the event of a crash
  - Can handle as many crashes as there are replicas of input data
  - Picks a new master if the master is lost
    - No single point of failure
- We demonstrate a proof of concept using 6 processes across 2 physical machines on a sample dataset

SQL Example

Dataset:

Rankings

Lists websites and their page rank:

Schema:

pageURL VARCHAR(300)

pageRank INT

avgDuration INT

Uservisits

Stores server logs for each web page

Schema:

sourceIP VARCHAR(116)

destURL VARCHAR(100)

adRevenue FLOAT

...*(ommitted)*

English Query:

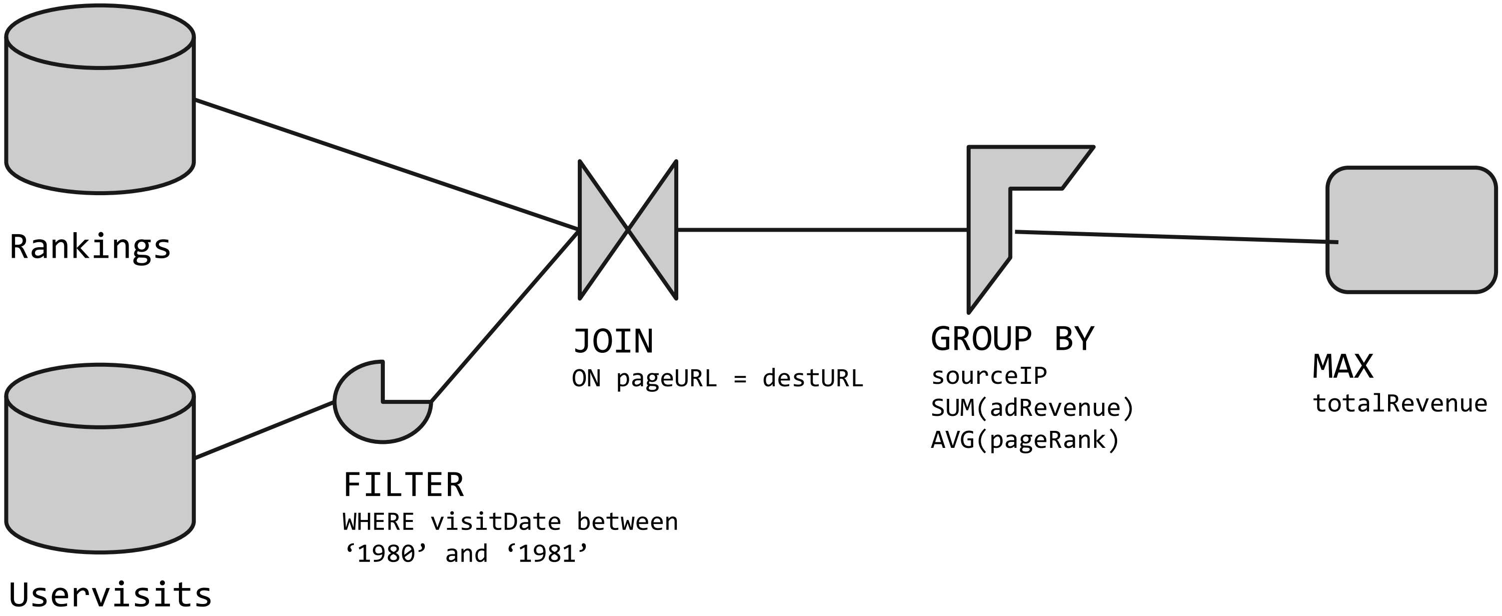
For the user that generated the most ad revenue during 1980:

- Report the sourceIP, total revenue, and average page rank of pages visited by this user

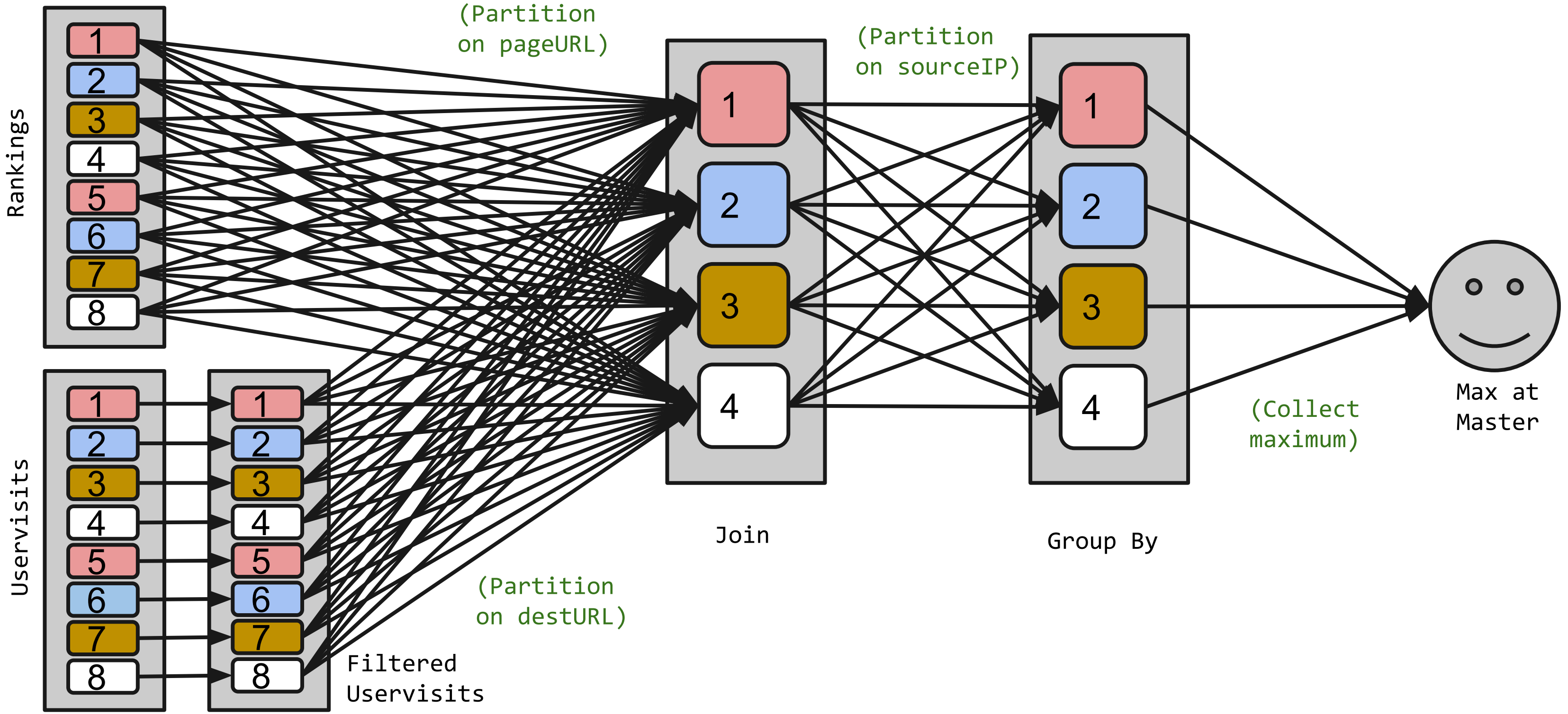
SQL Query:

```
SELECT sourceIP, totalRevenue, avgPageRank
FROM (SELECT sourceIP,
             AVG(pageRank) as avgPageRank,
             SUM(adRevenue) as totalRevenue
      FROM Rankings AS R, UserVisits AS UV
      WHERE R.pageURL = UV.destURL
        AND UV.visitDate BETWEEN Date(`1980-01-01)' AND  Date(`1981-01-01')
      GROUP BY UV.sourceIP)
ORDER BY totalRevenue DESC LIMIT 1
```

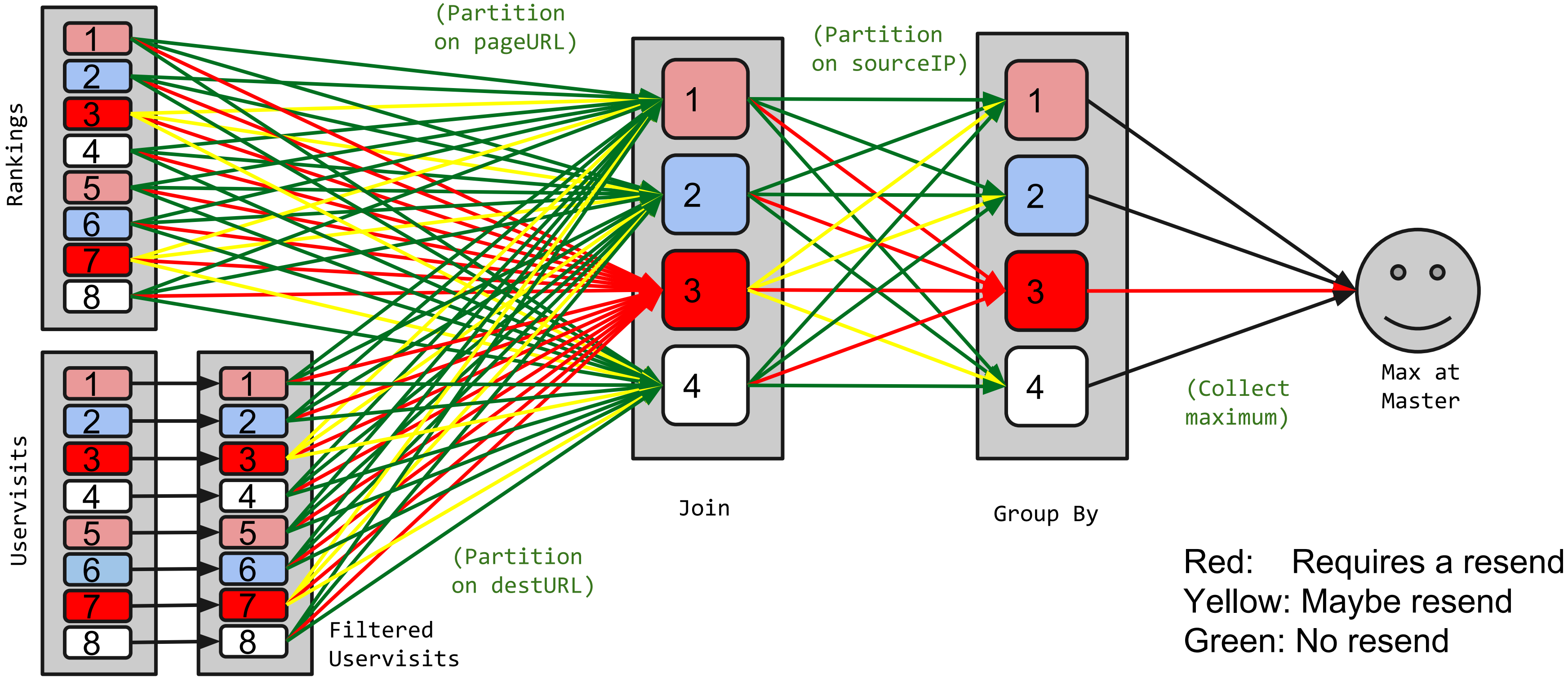
SQL Example: Logical Plan



SQL Example: Physical Plan



SQL Example: Gold Crashed



SQL Example: Implementation

- Assignment/Location of all partitions are known by all peers
  - Locations for replicas of input data are provided in deployment config
  - Assignments are a pure function of the current membership
- Request/Response Model
  - Request all dependencies required to perform local computation
  - When a request is received:
    - Compute locally and respond if all dependency data is local
      - Always possible at the leaves of the plan
    - Otherwise:
      - Request dependencies required for local computation
      - Respond after all requests are fulfilled
  - In the event of a membership change
    - Reassign all partitions. Reissue requests, as necessary

Demonstration

4 versions of the query:

- No Fault Tolerance *(gets stuck)*
- Terminate Gracefully
- Continue with missing data
- Replay missing work