Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems

Jonathan Mace, Ryan Roelke, Rodrigo Fonseca

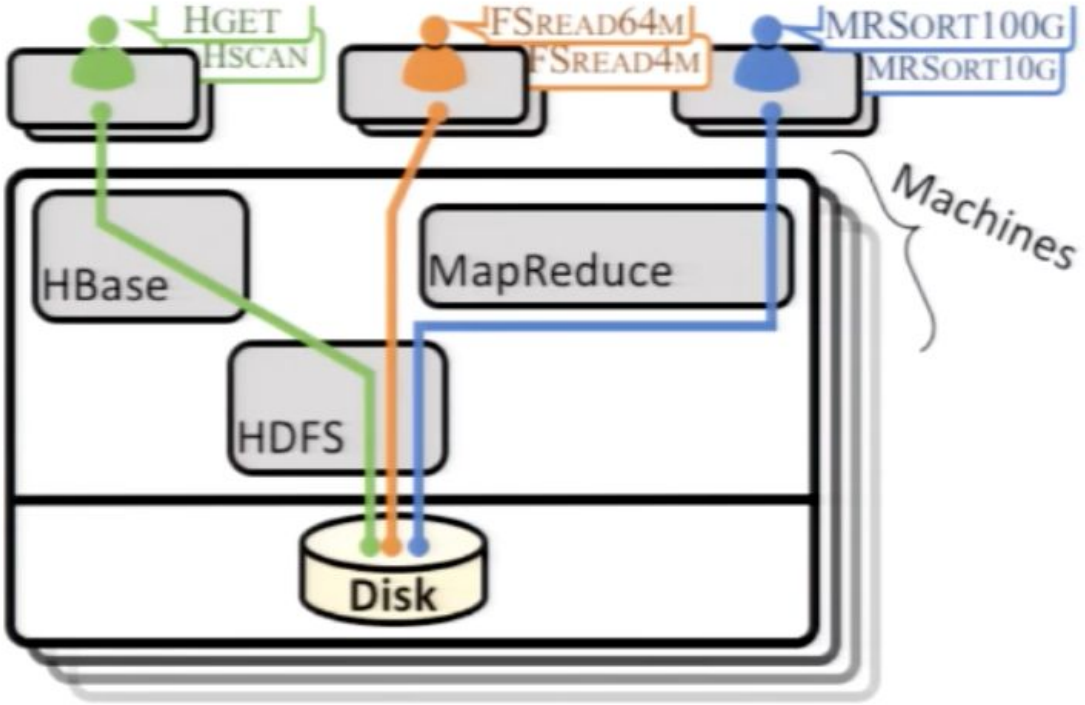
The problems with logging and monitoring

- problems in distributed systems are complex, varied, difficult to track and unpredictable
- the information needed to diagnose the problem is often not logged by the system
- if you need to get information that is not currently tracked by the system, you need to ask the developers to fix it or path the software yourself
- if you want to gather a lot of information and statistics, you do a lot of mess in the code, cause big overheads and generate tons of (irrelevant) messages which are then difficult to analyze
- there is a mismatch between the developer and user/operator needs
- simple logging and statistics can’t track issues after they cross current process/system/machine boundary

Pivot tracing

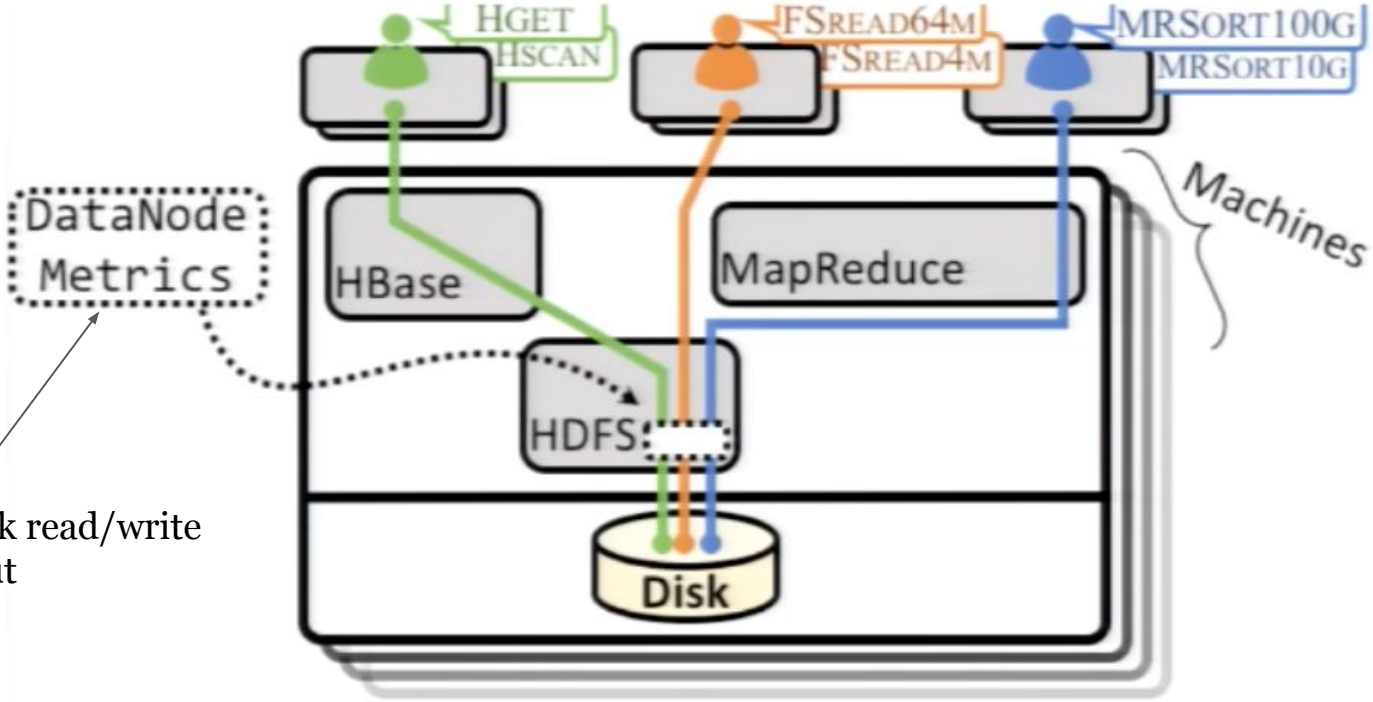
- dynamically install and configure monitoring at runtime
- low system overhead to allow ‘always on’ monitoring
- capture causality between from multiple processes and applications

Tracking disk bandwidth



Tracking disk bandwidth

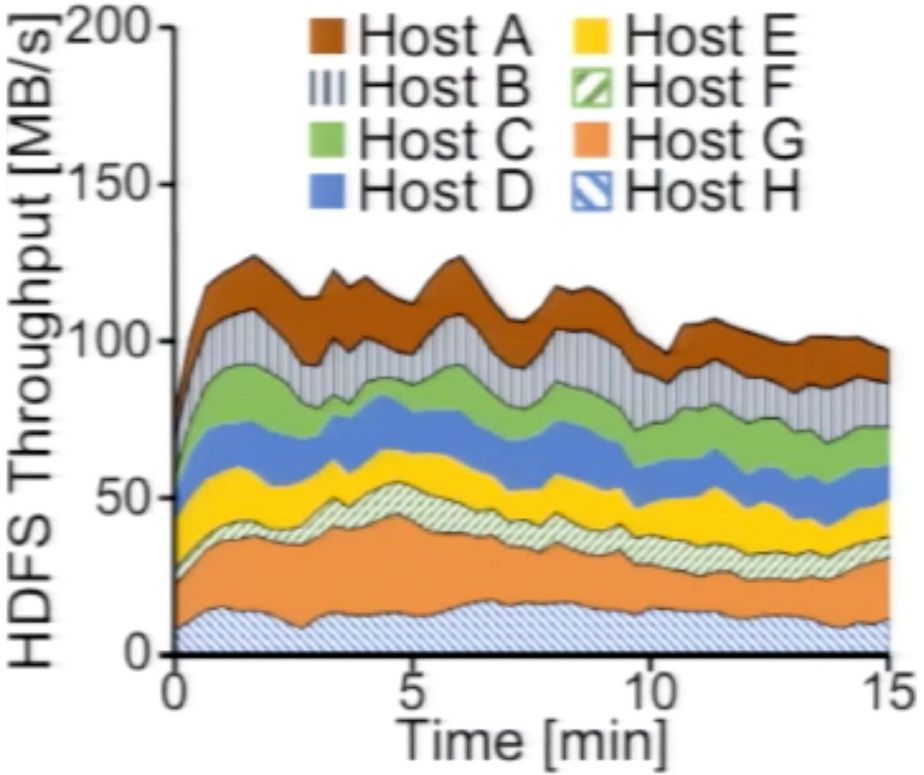
- at HDFS



Tracks disk read/write throughput

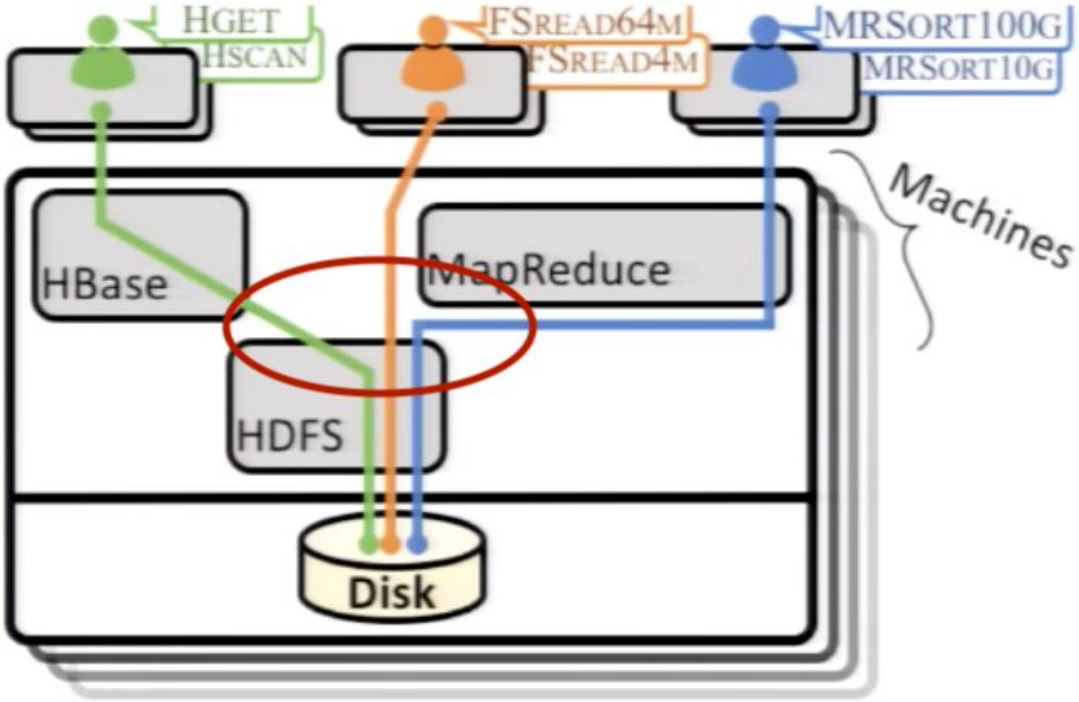
Tracking disk bandwidth

- at HDFS



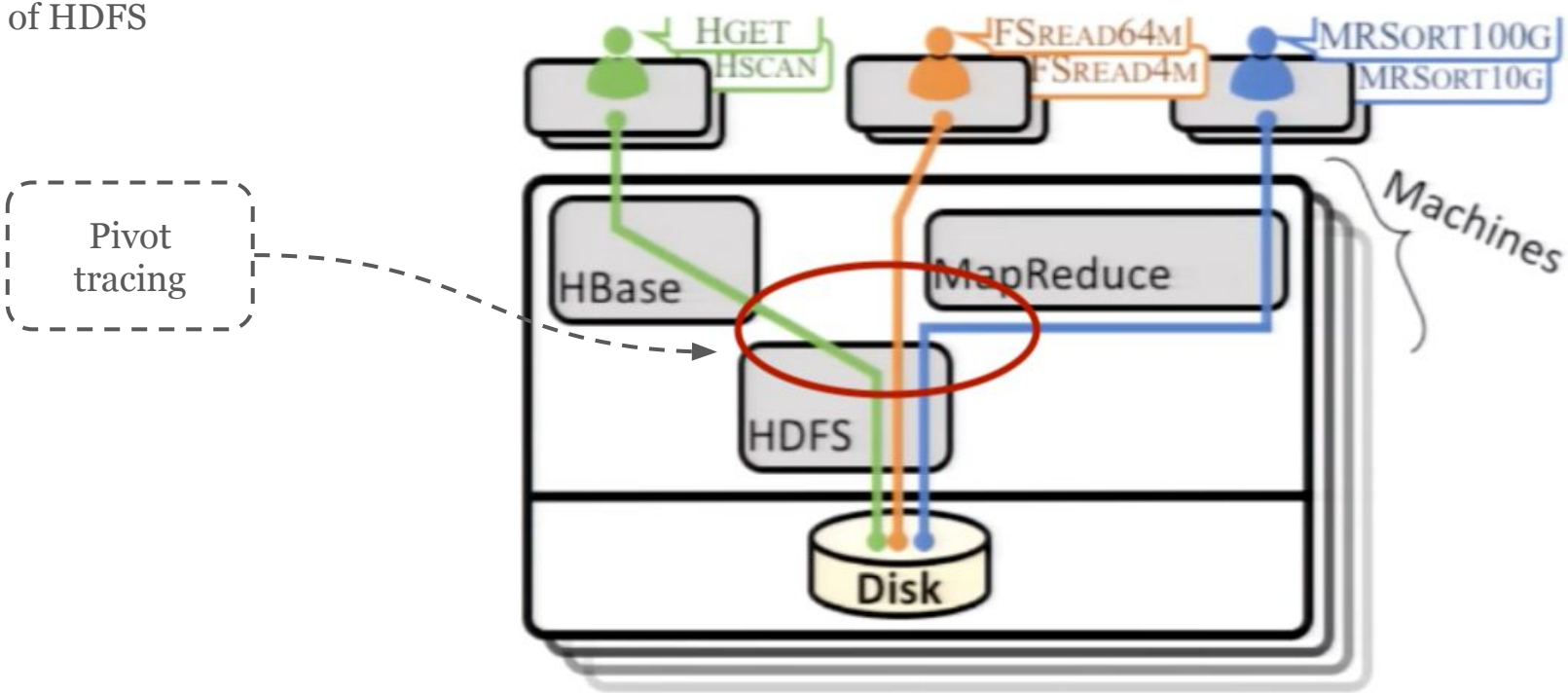
Tracking disk bandwidth

- at application on top of HDFS



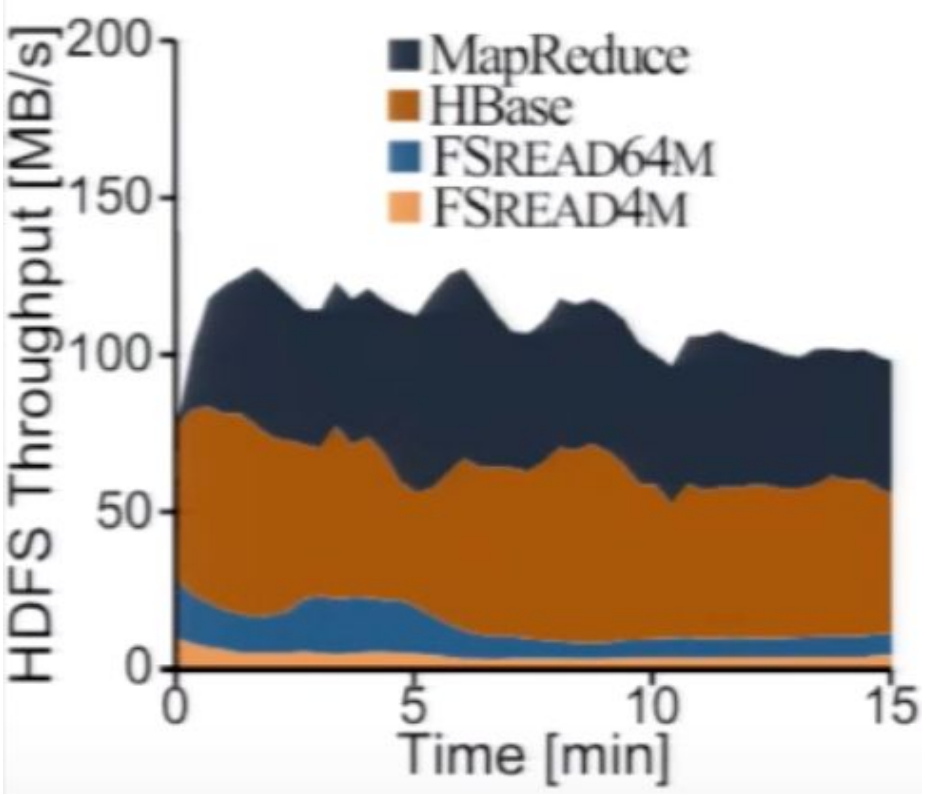
Tracking disk bandwidth

- at application on top of HDFS



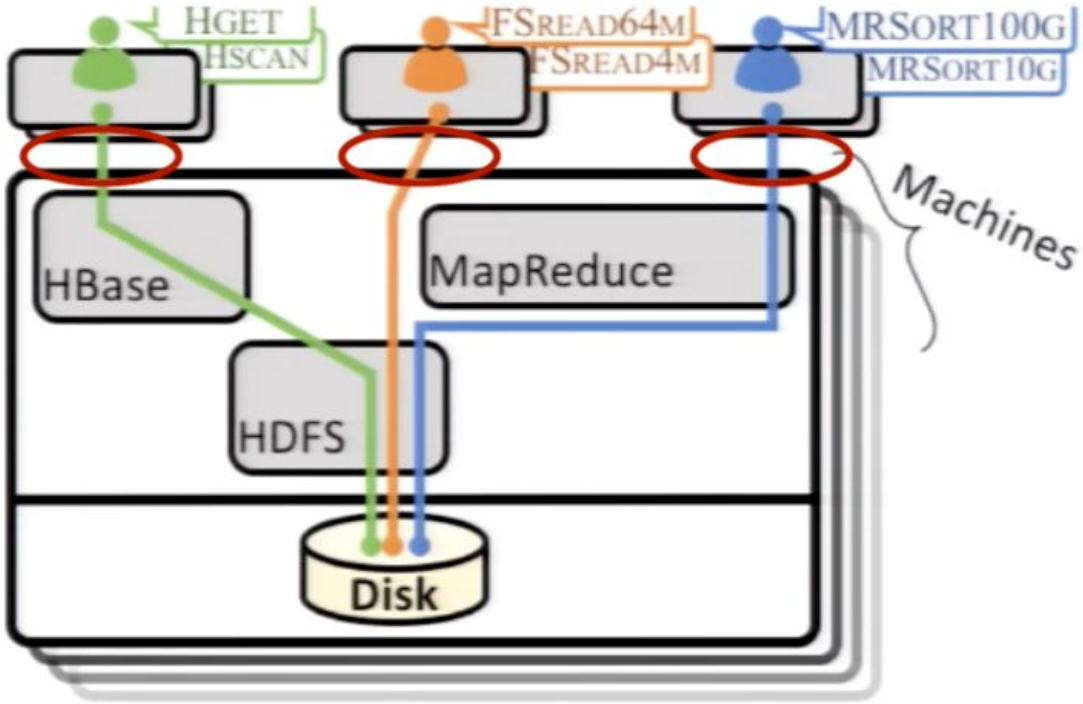
Tracking disk bandwidth

- at application on top of HDFS



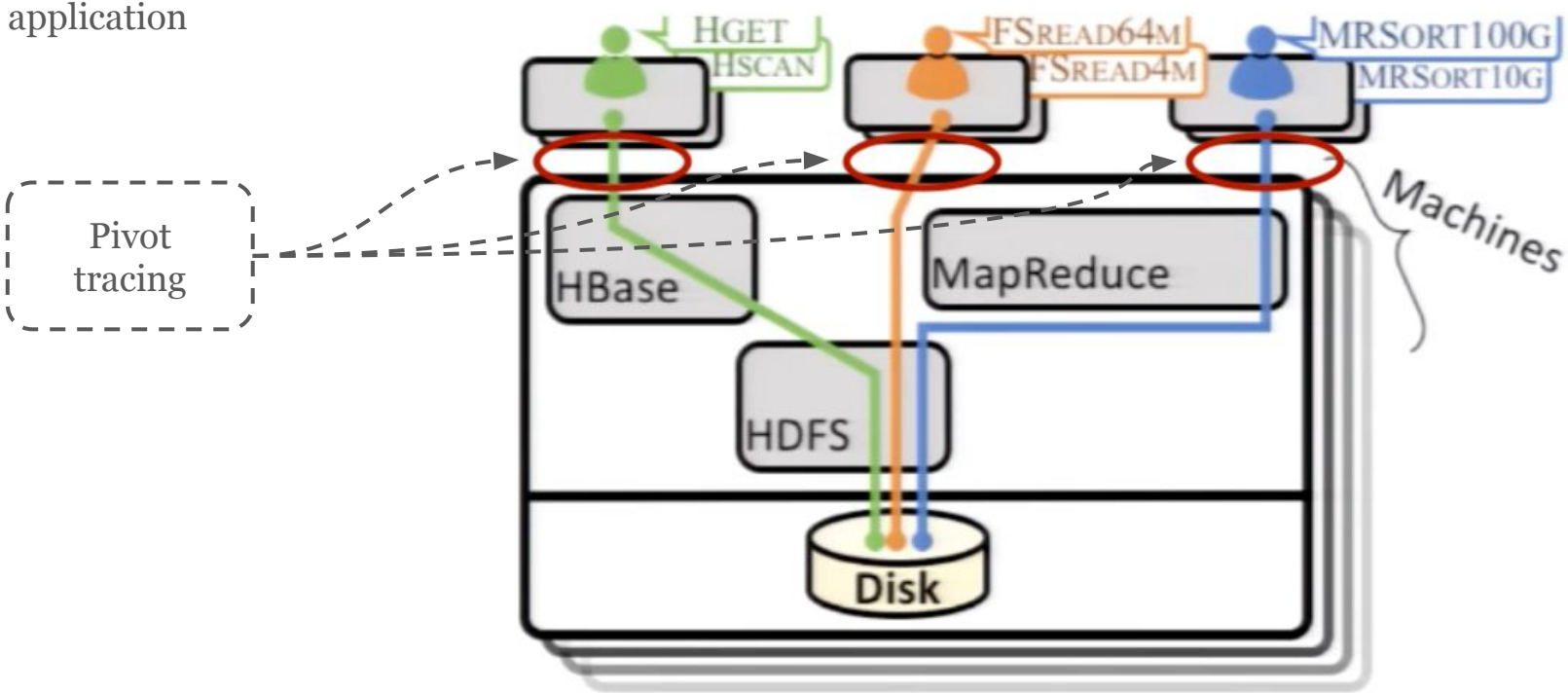
Tracking disk bandwidth

- at top level user application



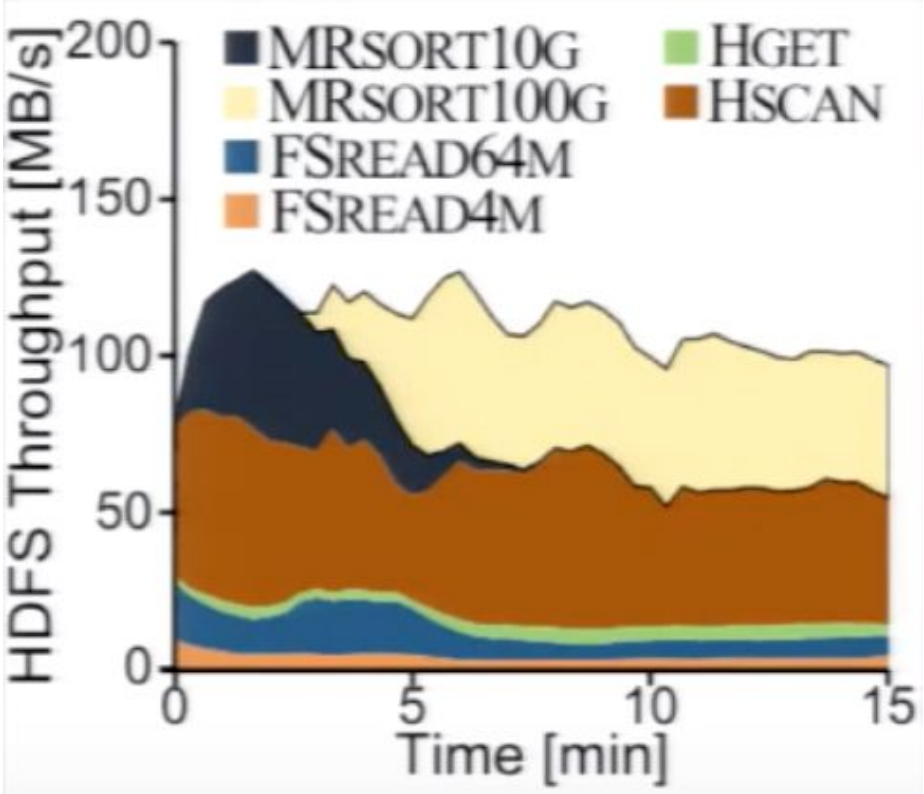
Tracking disk bandwidth

- at top level user application



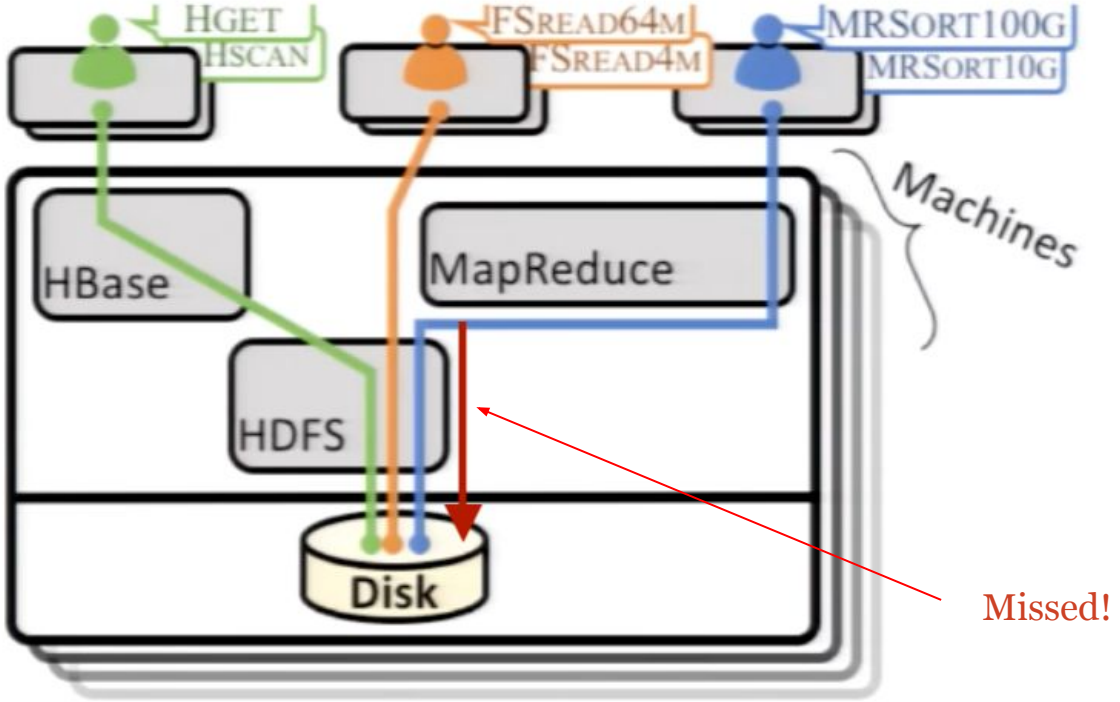
Tracking disk bandwidth

- at top level user application



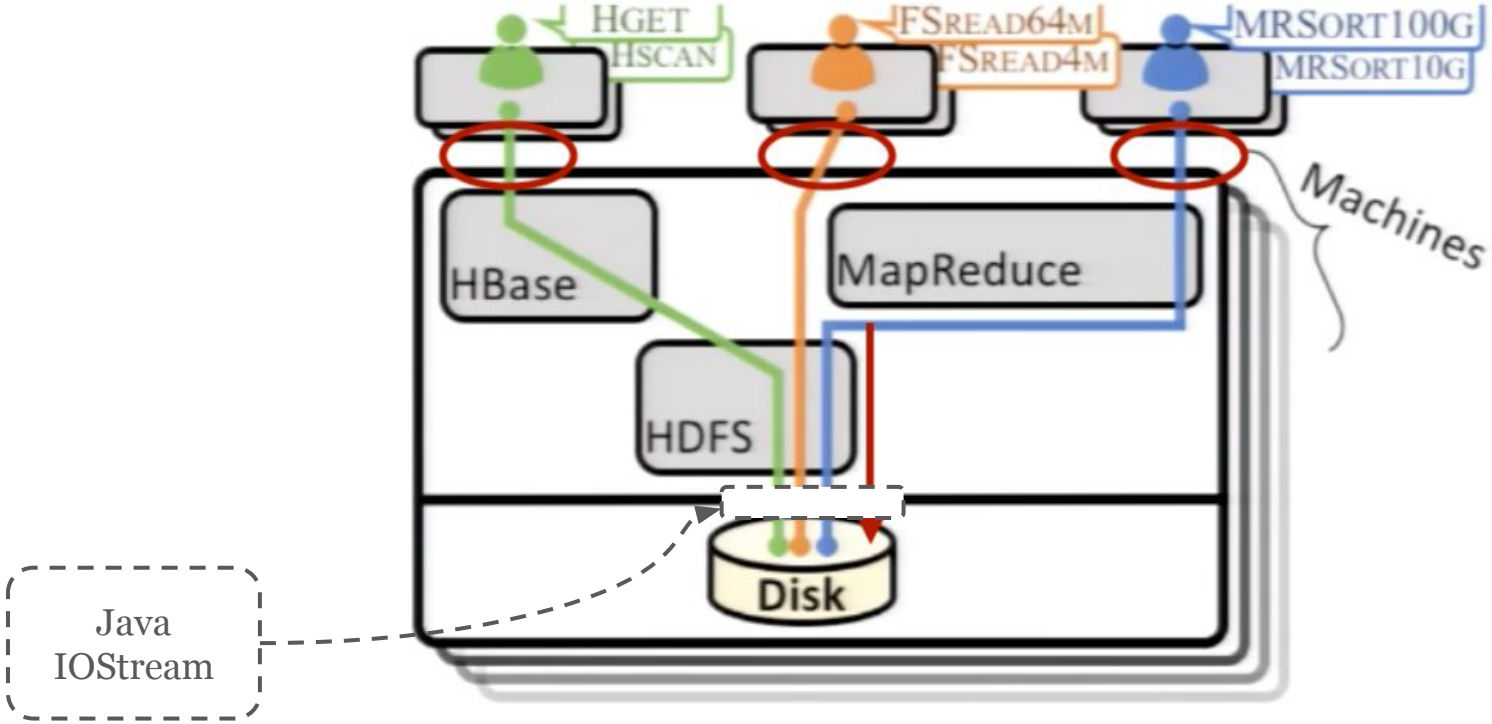
Tracking disk bandwidth

- other ways?



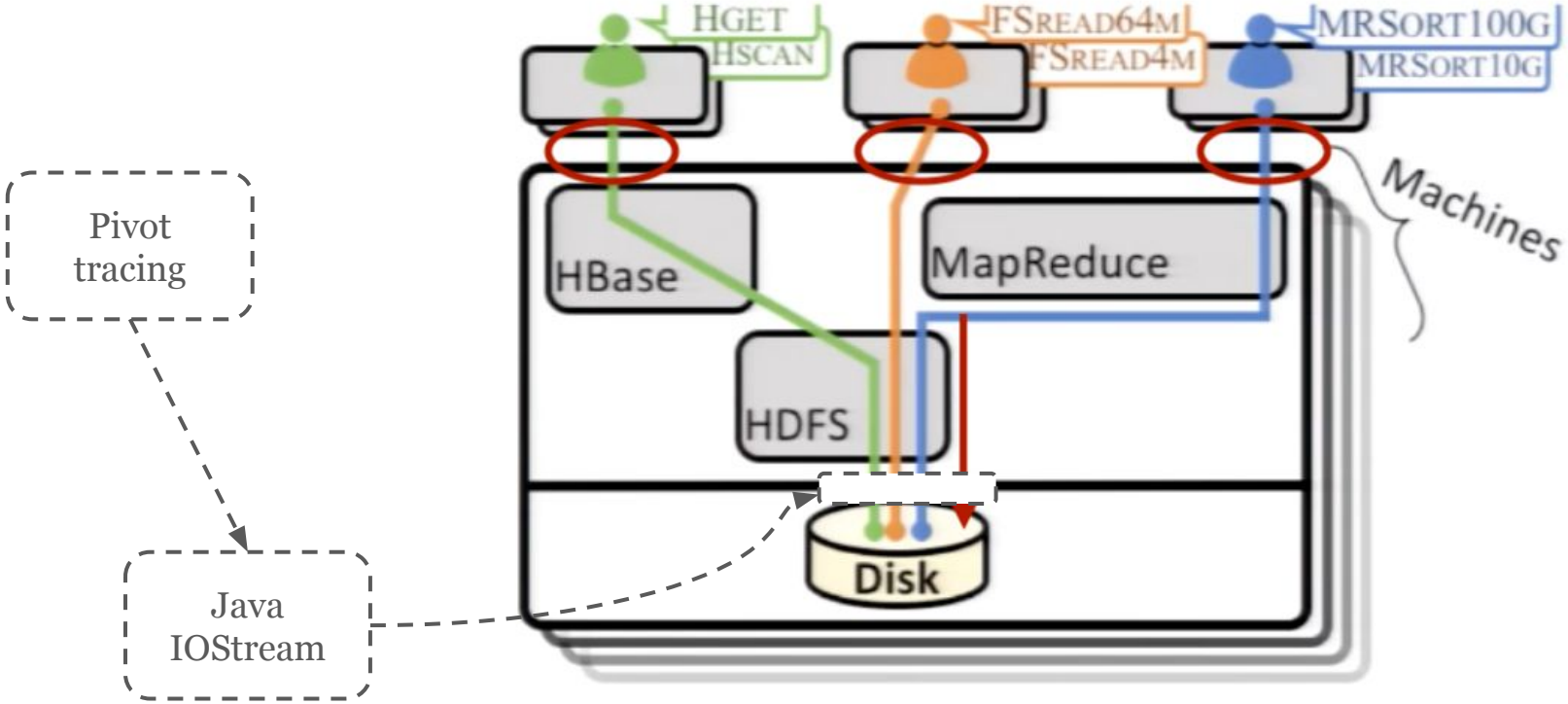
Tracking disk bandwidth

- other ways



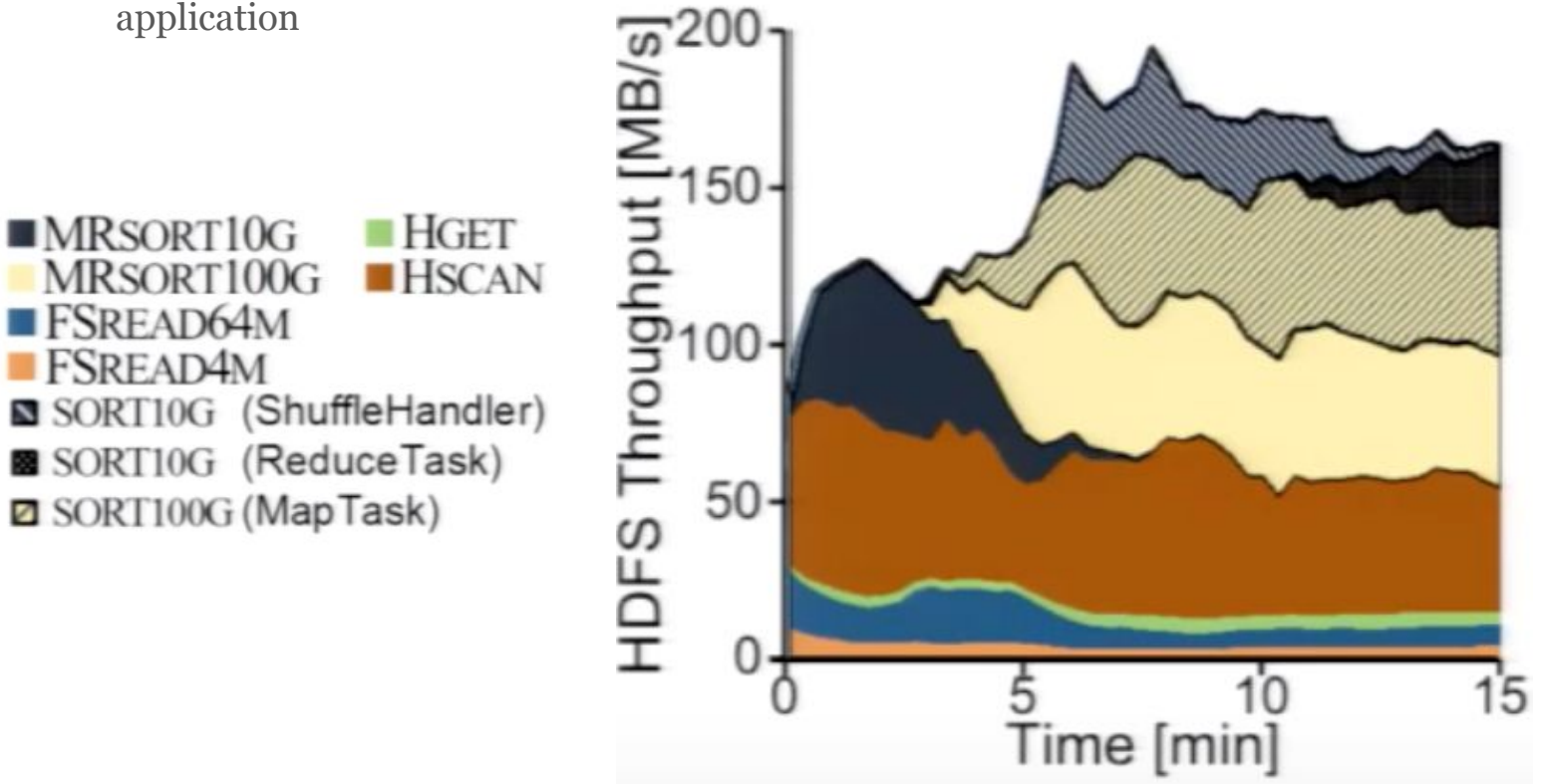
Tracking disk bandwidth

- other ways



Tracking disk bandwidth

- at top level user application



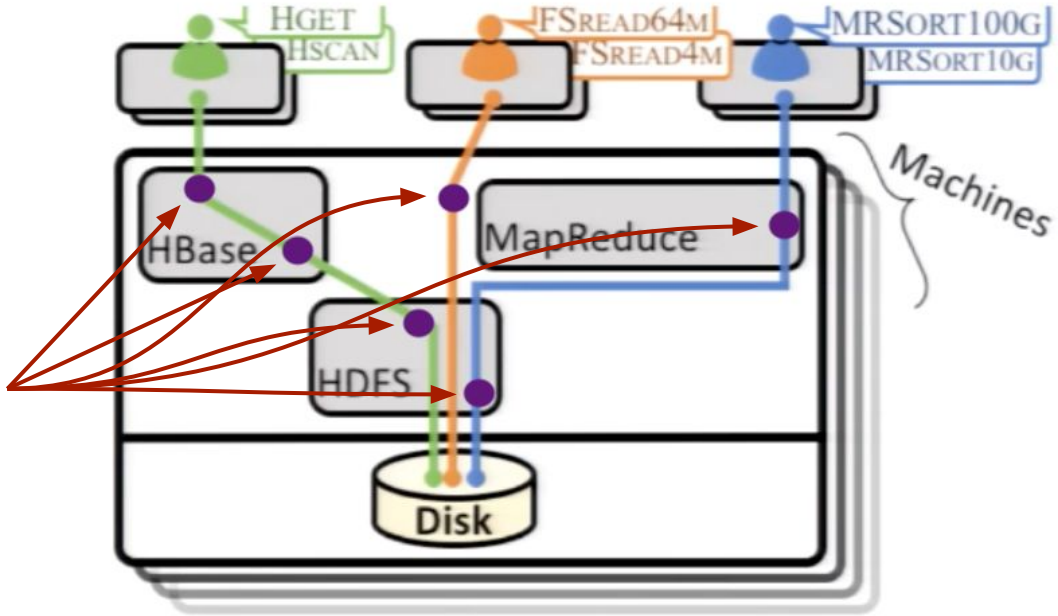
Pivot tracing

Overview

Data set

Model system events as a streaming, distributed data set of tuples and dynamically evaluate relational queries over this dataset.

Events
(e.g. some IO action, receiving a request etc.)

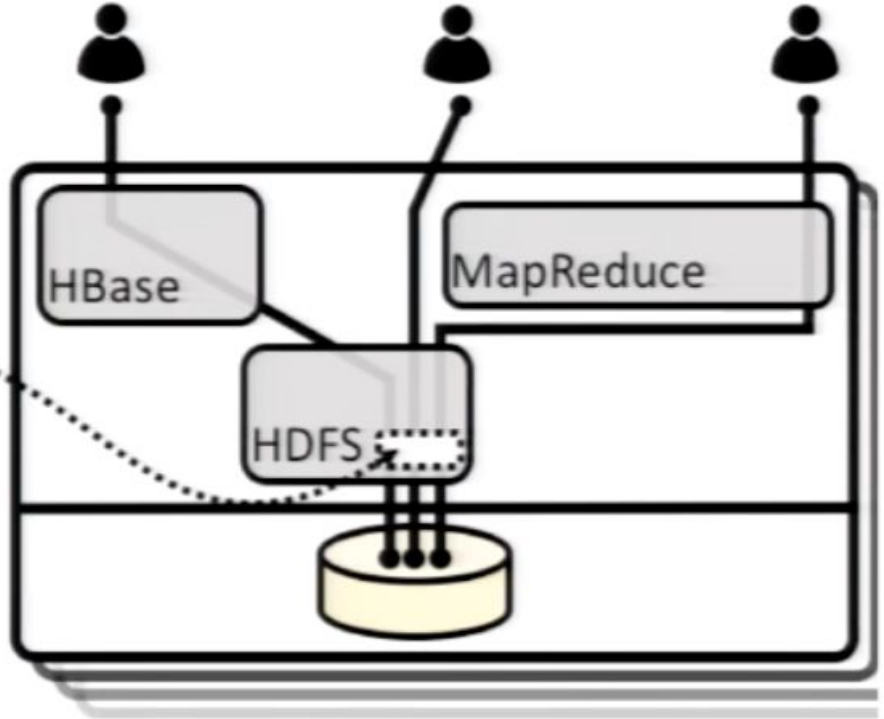


Events

DataNodeMetrics.java

```
50   public class DataNodeMetrics {
       ...
266    public void incrBytesRead(int delta) {
267      ...
268    }
       ...
407  }
```

DataNode Metrics

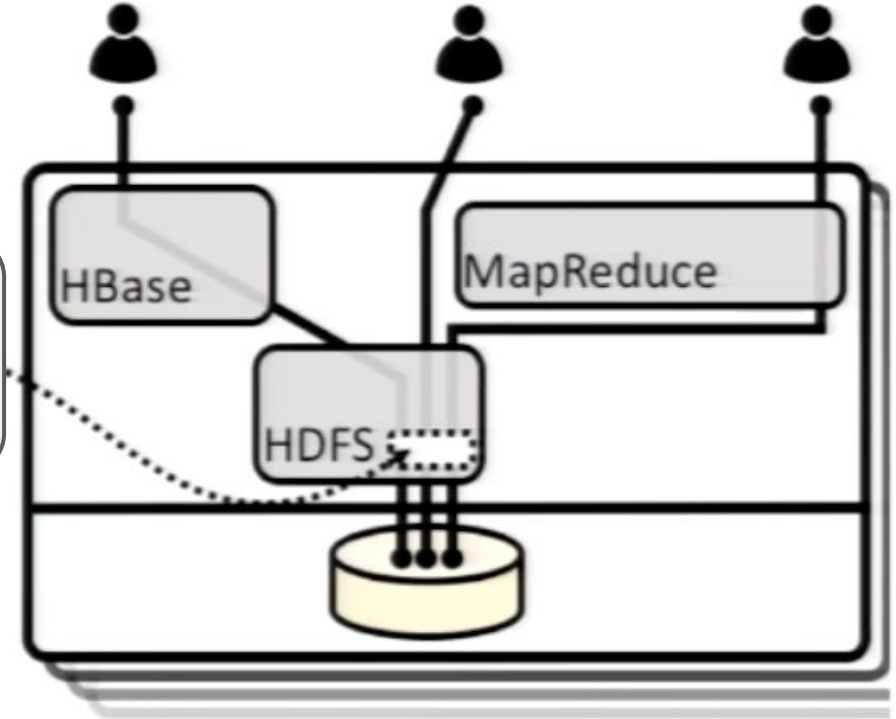


Tracepoints

DataNodeMetrics.java

```
50   public class DataNodeMetrics {
       ...
266    public void incrBytesRead(int delta) {
267      ...
268    }
       ...
407  }
```

Tracepoint
Class: DataNodeMetrics
Method: incrBytesRead
Exports: ‘delta’=delta



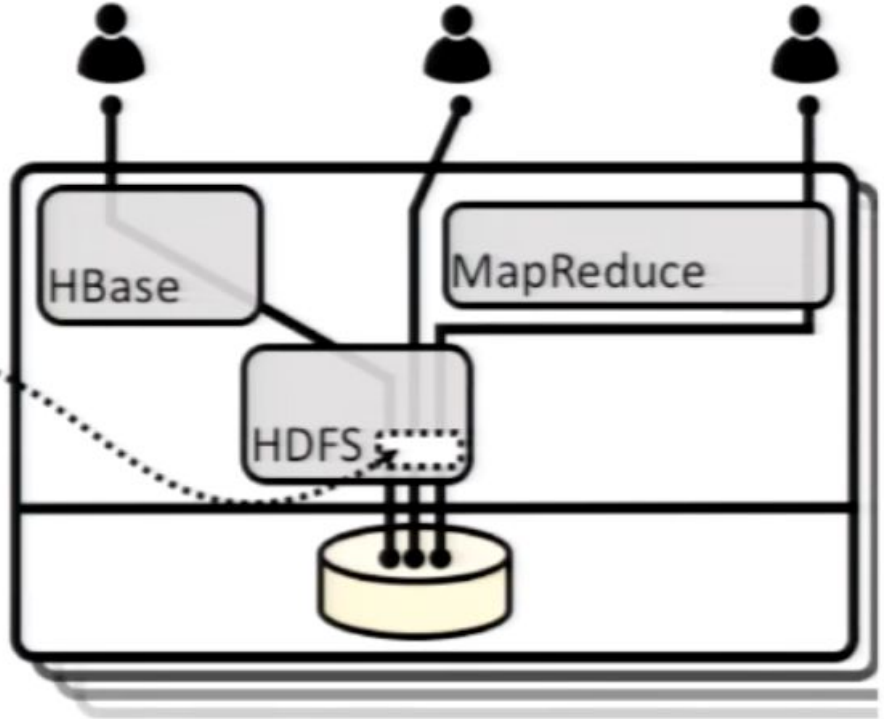
Events and tuples

DataNodeMetrics.java

```
50   public class DataNodeMetrics {
       ...
266    public void incrBytesRead(int delta) {
267      ...
268    }
       ...
407  }
```

Tracepoint
Class: DataNodeMetrics
Method: incrBytesRead
Exports: ‘delta’=delta

(“DataNodeMetrics”, delta=10, host=”host1”, …)

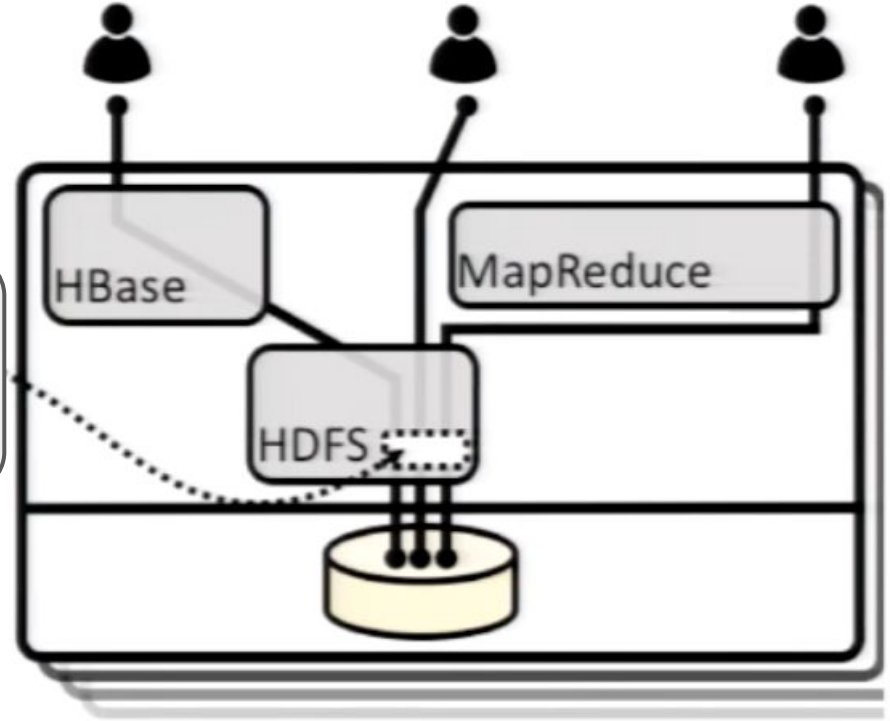


Query Language

Operation	*Description*	*Example*
From	Use input tuples from a set of tracepoints	`From e In RPCs`
Union (⋃)	Union events from multiple tracepoints	`From e In DataRPCs, ControlRPCs`
Selection (σ)	Filter only tuples that match a predicate	`Where e.Size < 10`
Projection (Π)	Restrict tuples to a subset of fields	`Select e.User, e.Host`
Aggregation (A)	Aggregate tuples	`Select SUM(e.Cost)`
GroupBy (G)	Group tuples based on one or more fields	`GroupBy e.User`
GroupBy Aggregation (GA)	Aggregate tuples of a group	`Select e.User, SUM(e.Cost)`
Happened-Before Join ($\overrightarrow{\bowtie}$)	Happened-before join tuples from another query	`Join d In Disk On d -> e`
	Happened-before join a subset of tuples	`Join d In MostRecent(Disk) On d -> e`

Queries on tuples

Tracepoint
Class: DataNodeMetrics
Method: incrBytesRead
Exports: ‘delta’=delta

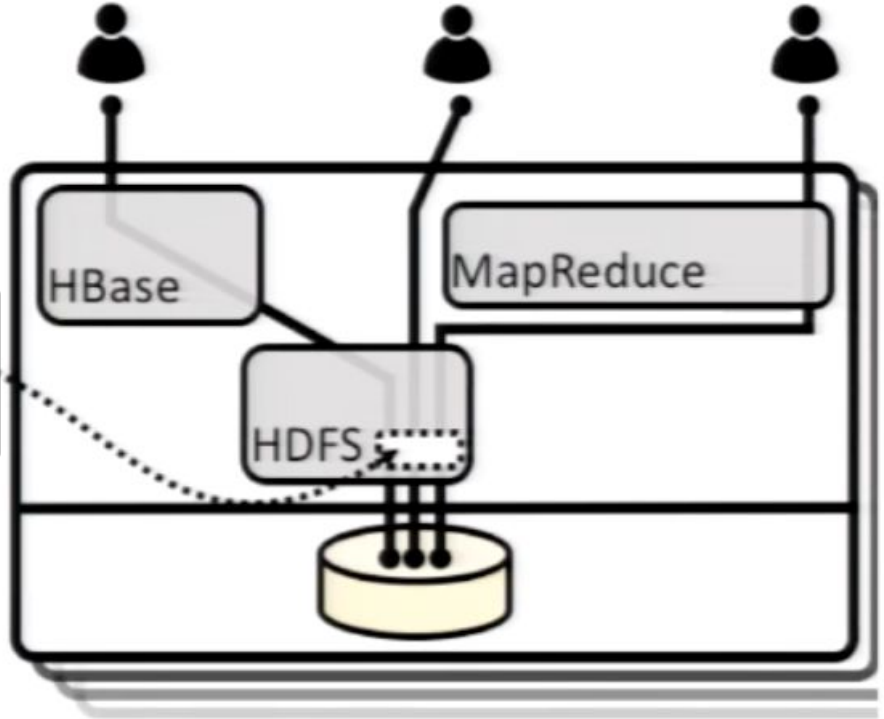


(“DataNodeMetrics”, delta=10, host=”host1”, …)

Queries on the tuples

```
From incr In DataNodeMetrics.incrBytesRead
     GroupBy incr.host
     Select incr.host, SUM(incr.delta)
```

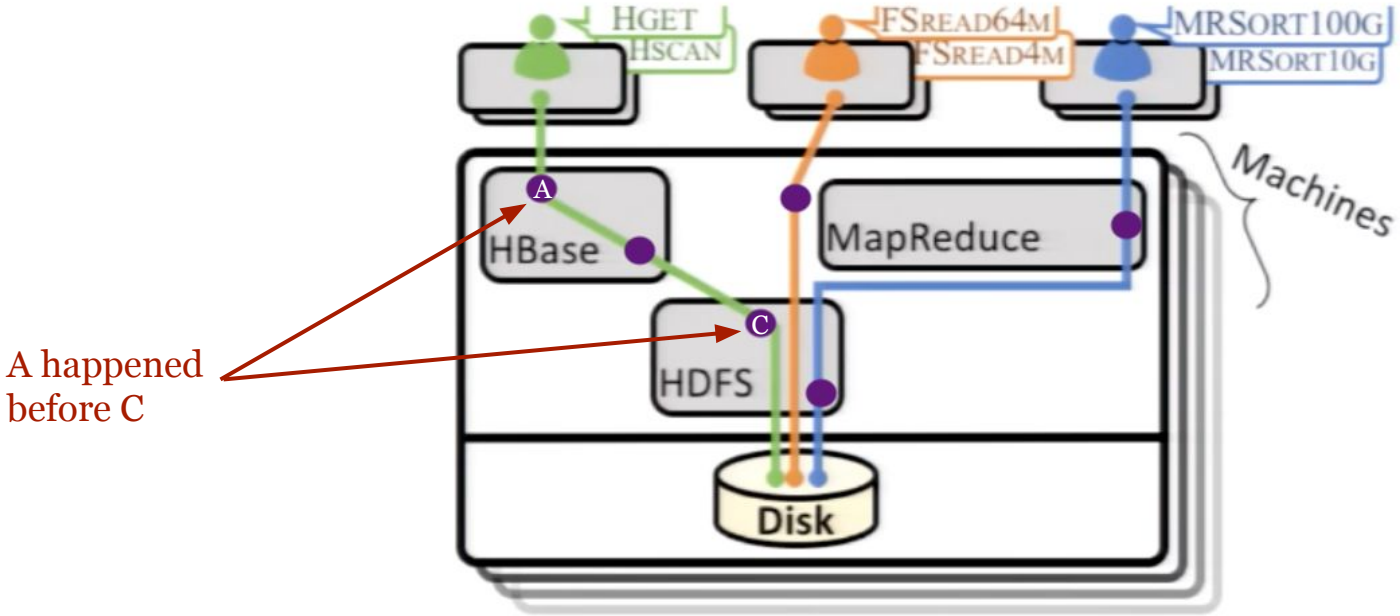
Tracepoint
Class: DataNodeMetrics
Method: incrBytesRead
Exports: ‘delta’=delta



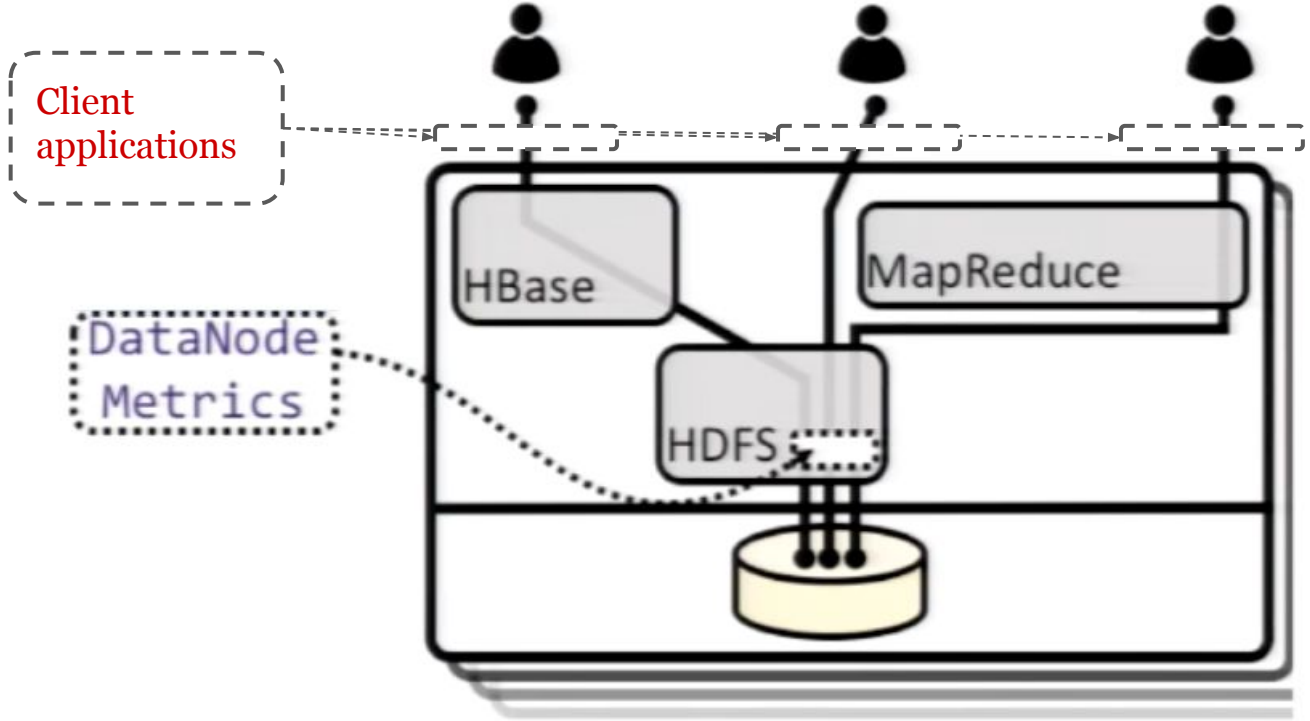
(“DataNodeMetrics”, delta=10, host=”host1”, …)

Causal tracing

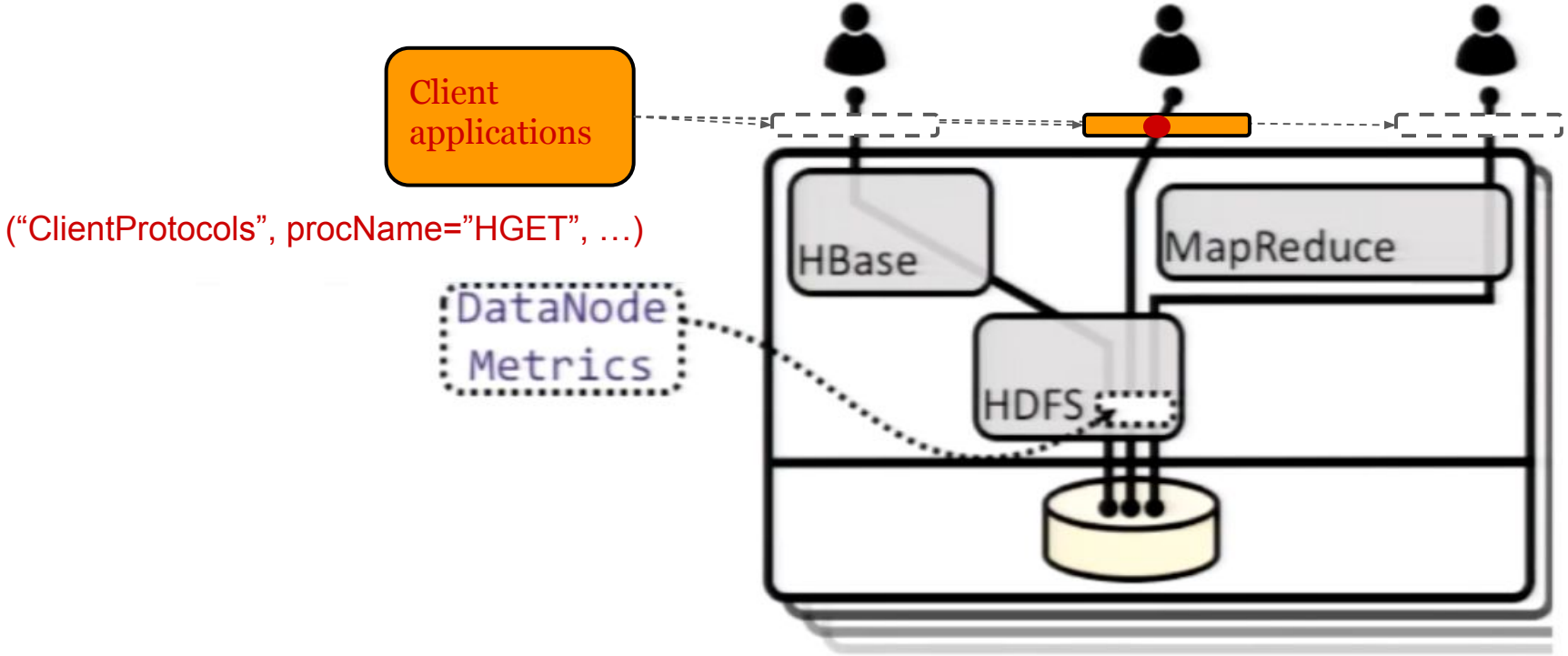
Happened-before join ($\overrightarrow{\bowtie}$) operation



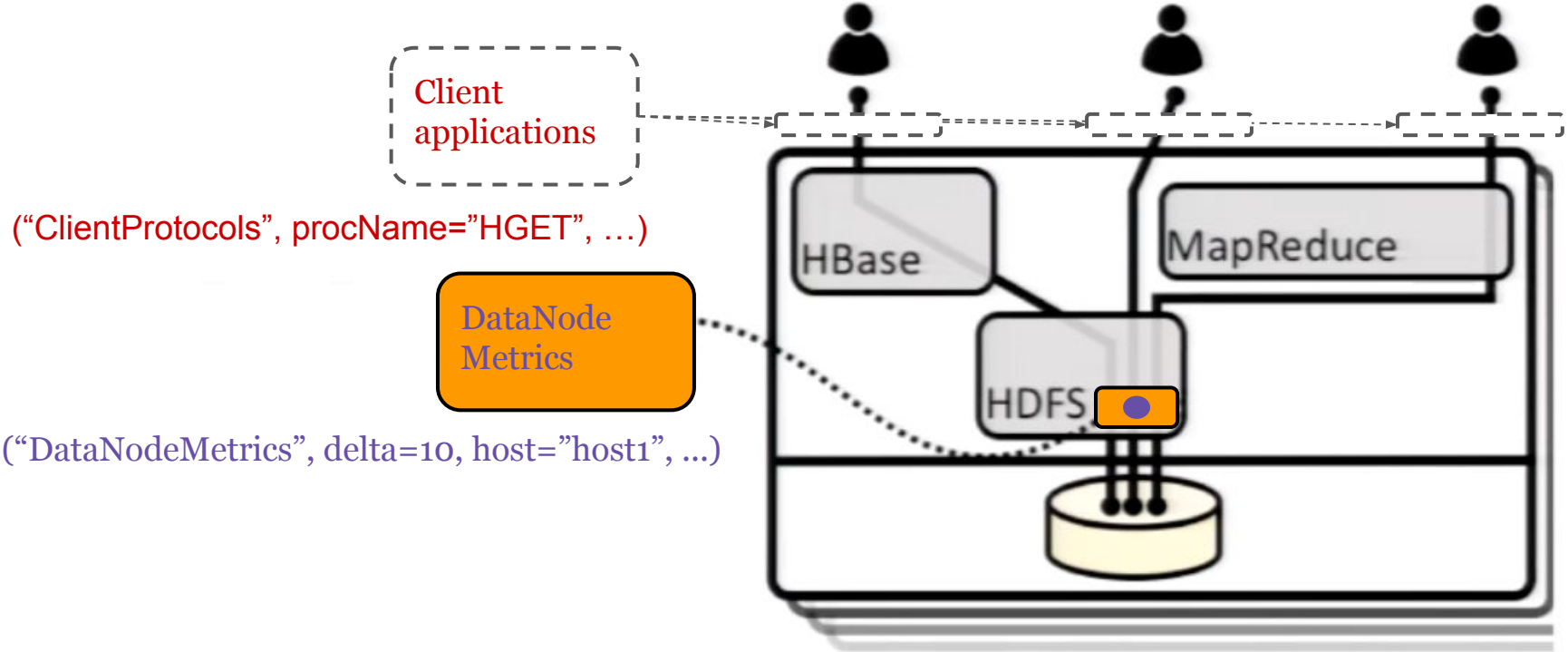
Causal tracing



Causal tracing



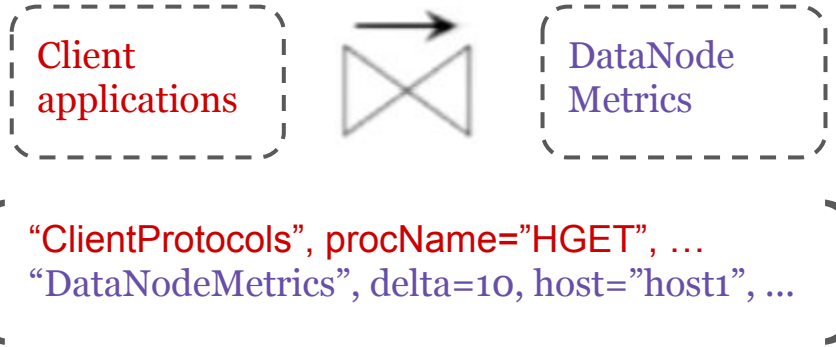
Causal tracing



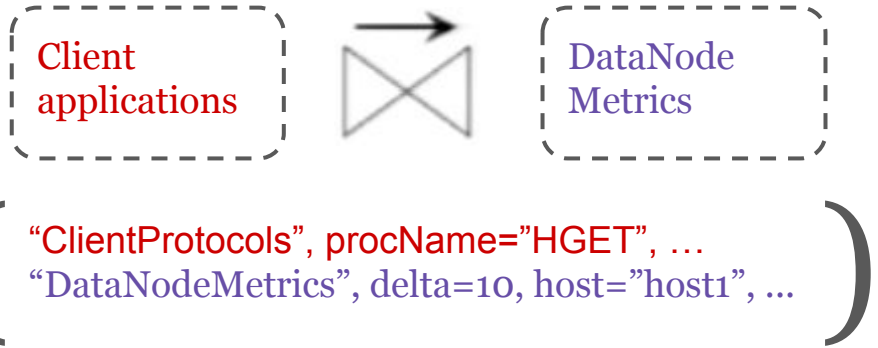
Happened-before join 

Client applications  DataNode Metrics

Happened-before join 

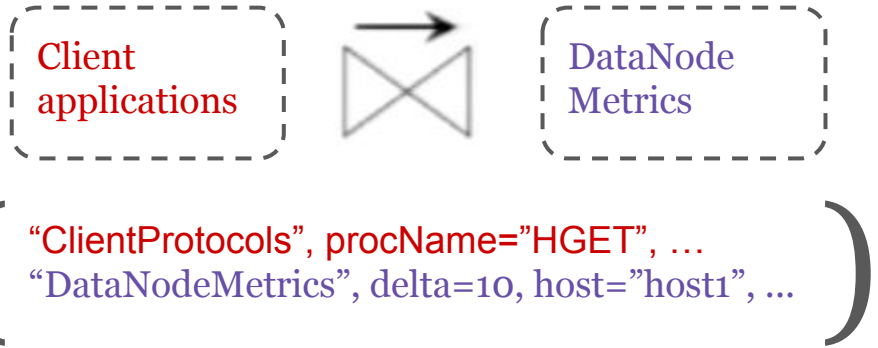


Happened-before join 




```
From incr In DataNodeMetrics.incrBytesRead
    Join client In First(ClientProtocols) On client -> incr
    GroupBy client.procName
    Select client.procName, SUM(incr.delta)
```

Happened-before join 




```
From incr In DataNodeMetrics.incrBytesRead
    Join client In First(ClientProtocols) On client -> incr
    GroupBy client.procName
    Select client.procName, SUM(incr.delta)
```

(procName="HGET", delta=10)

Design and implementation

Pre-requisites

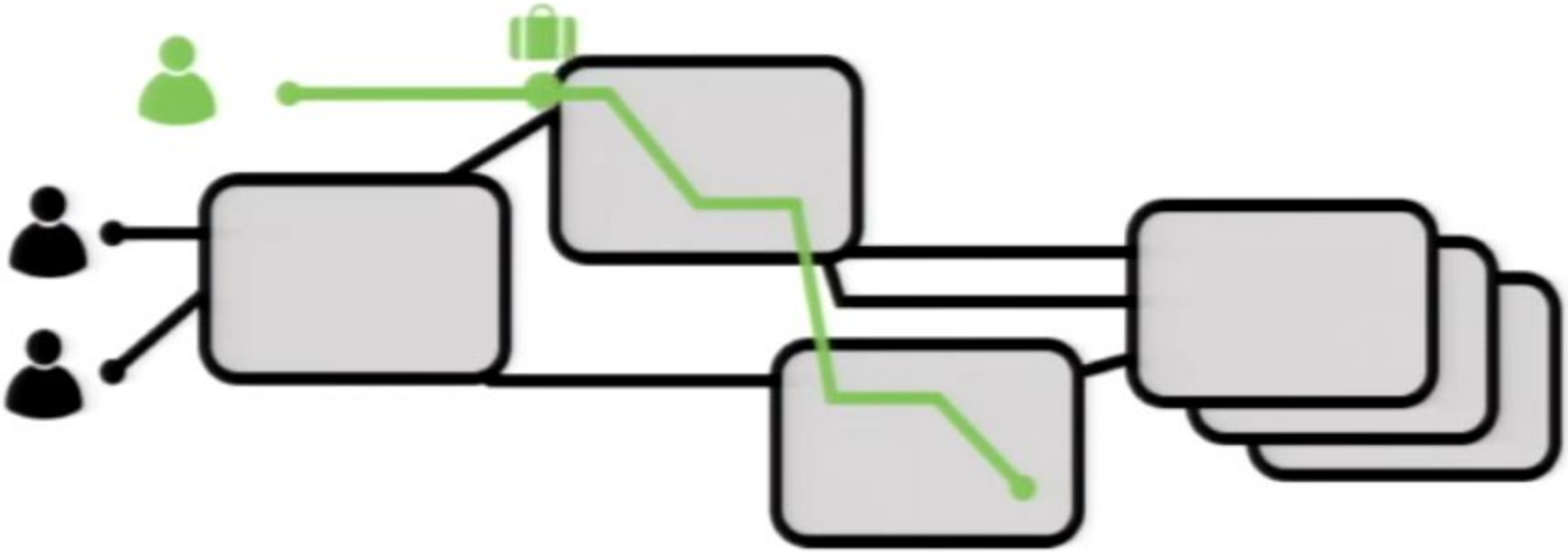
Dynamic instrumentation

Queries are dynamically installed into the system at runtime using dynamic instrumentation. In order to do be able to do this, the system needs to install PT Agents.

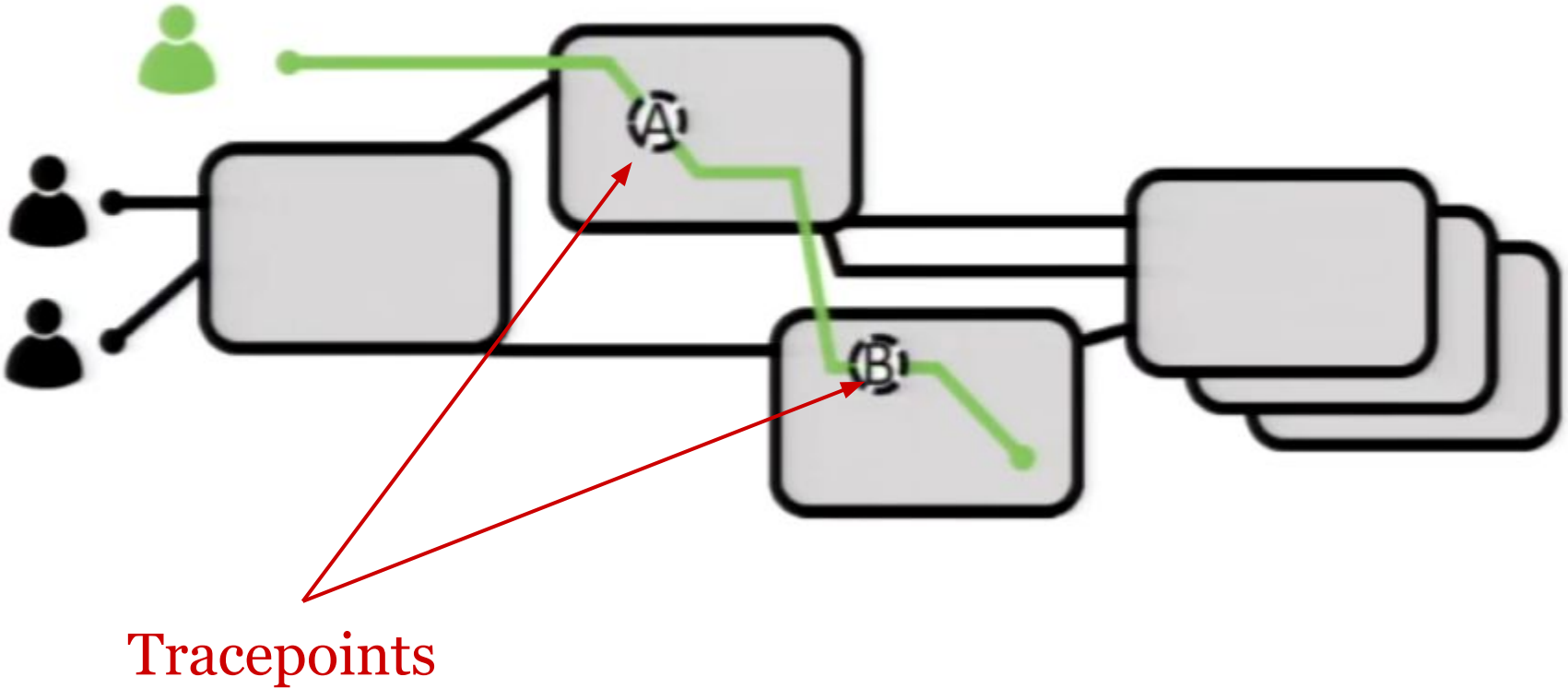
Causal tracing

In order to be able to perform causal tracing, the system needs to use Baggage abstraction. It makes the implementation of happened-before join possible.

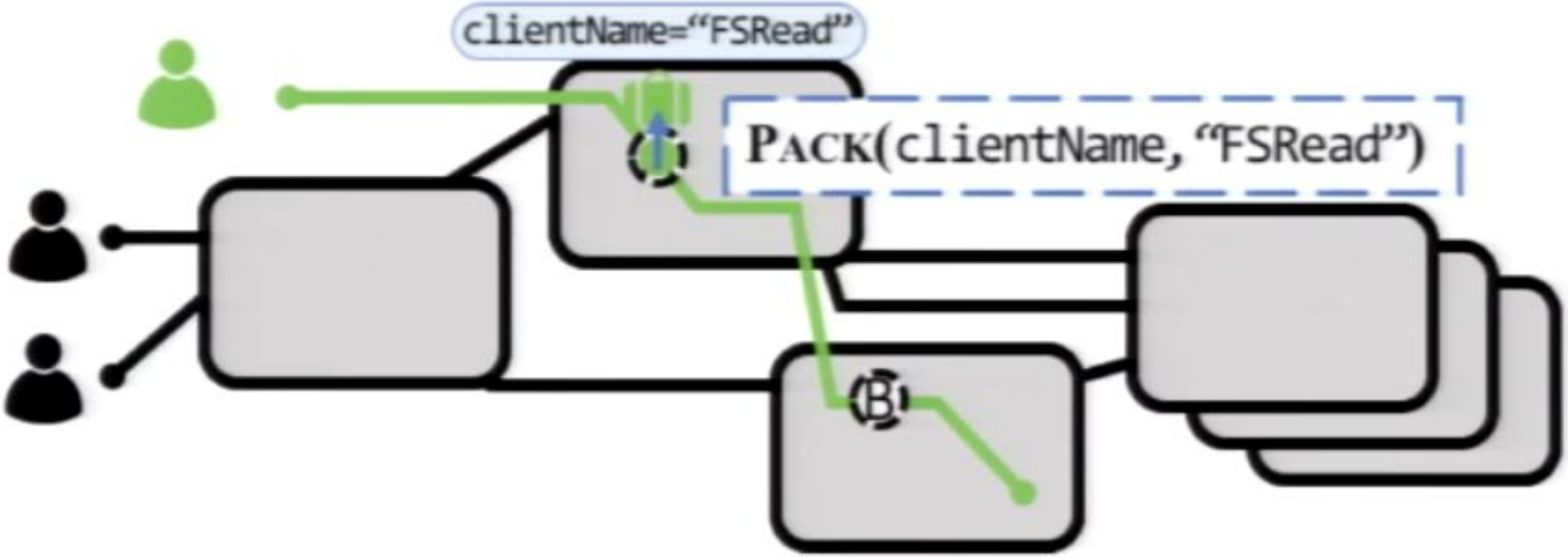
Baggage



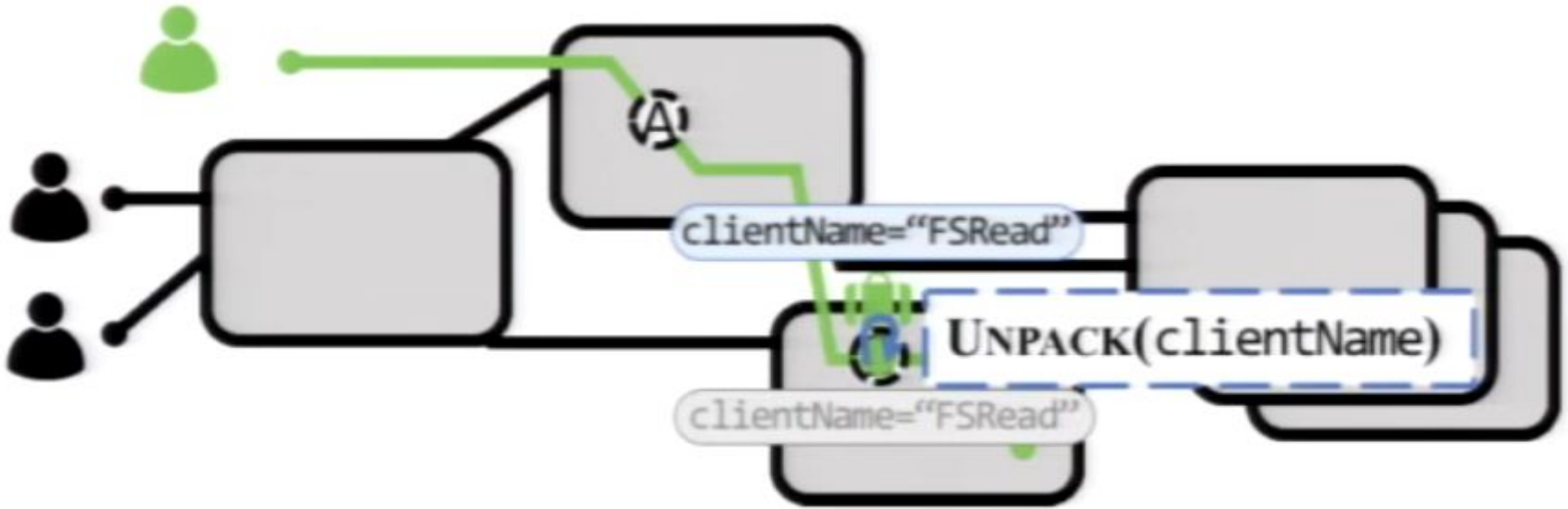
Baggage



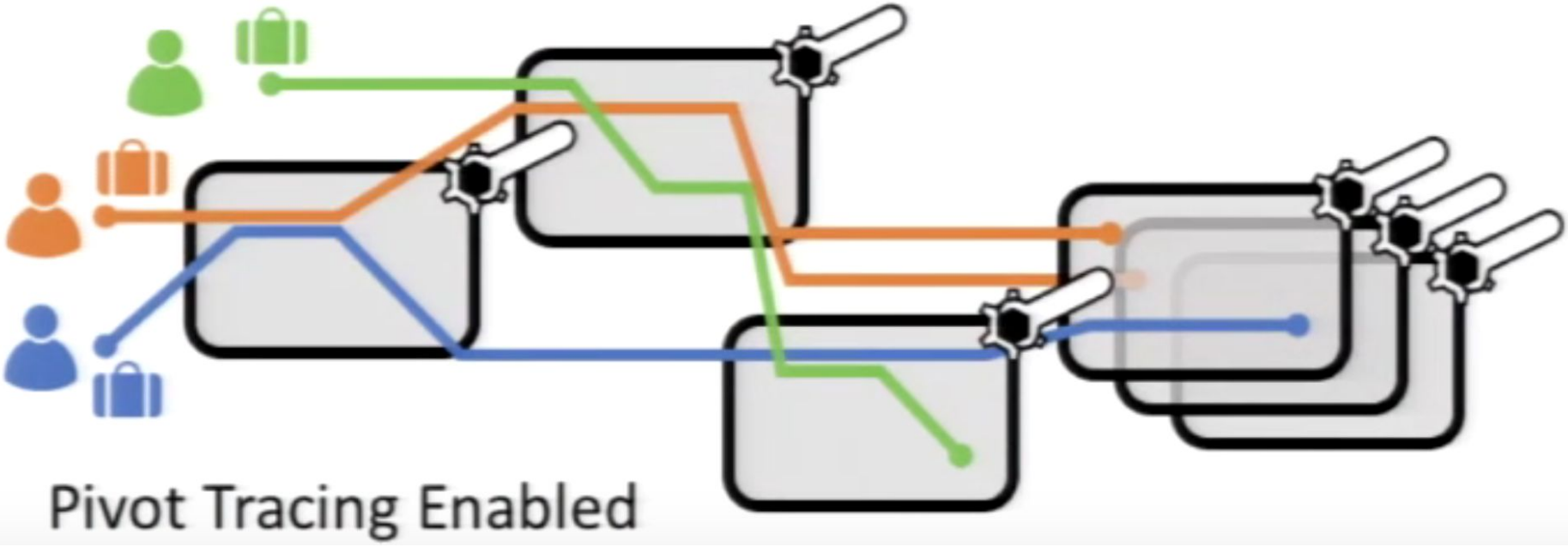
Baggage



Baggage



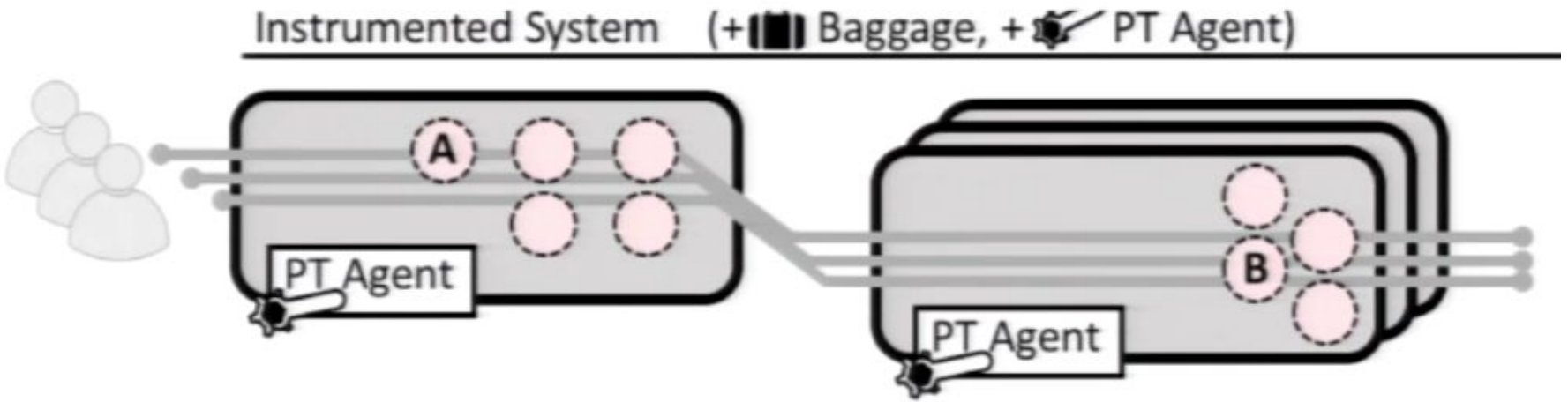
Baggage + PT Agents



Design and implementation

Queries

Tracepoints



Tracepoints are places where Pivot Tracing can act. They export identifiers that will be accessible to queries.
By default each tracepoint exports host, timestamp, pid and proc name.

Tracepoint A
Class: A
Method: A1()

Tracepoint B
Class: B
Method: B1()
Exports: ‘delta’=delta

Queries

Queries can refer to identifiers exported by tracepoints.

The output of a query is a stream of tuples (e.g. 10 tuples every 1s) - for example (procName, delta).

Advices

When user submits the query to the Pivot Tracing frontend it is compiled to an advice.

Advices are an intermediate representation that will be installed and used at tracepoints. They specify what operations to perform at each tracepoint used in the query.

They have a limited instruction set: OBSERVE, PACK, FILTER, UNPACK, EMIT.

Advice code is restricted: it has no jumps, no recursion and is guaranteed to terminate.

Query compilation example

```
From incr In DataNodeMetrics.incrBytesRead
    Join client In First(ClientProtocols)
On client -> incr
    GroupBy client.procName
    Select client.procName,
SUM(incr.delta)
```

A1

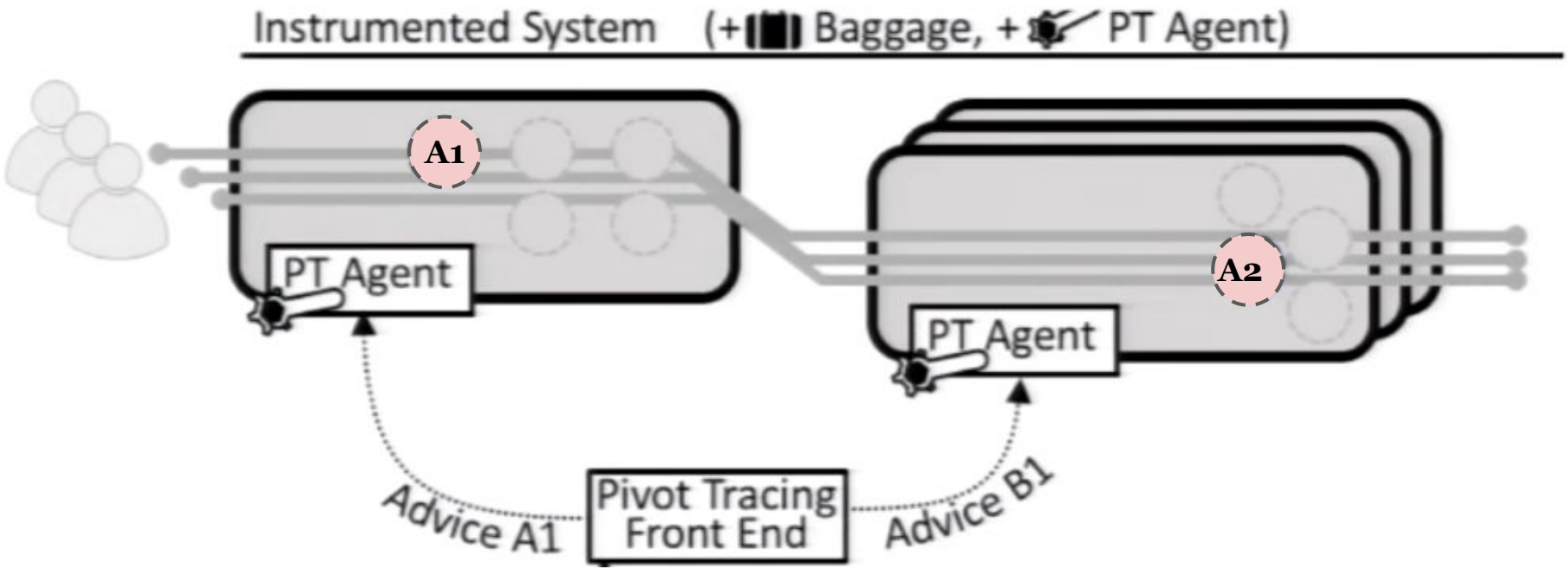
Advice A1
OBSERVE procName
PACK procName

A2

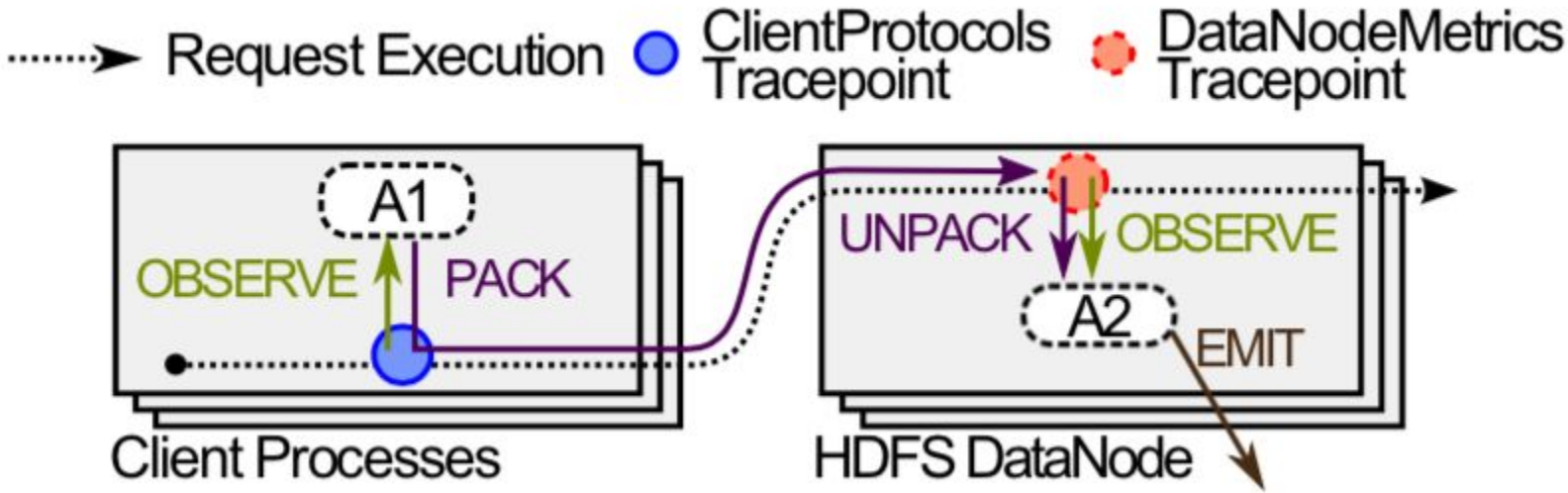
Advice A2
OBSERVE delta
UNPACK procName
EMIT procName, SUM(delta)

Tracepoint
Class: DataNodeMetrics
Method: incrBytesRead
Exports: ‘delta’=delta

Adding advice to the tracepoints



Advice execution example



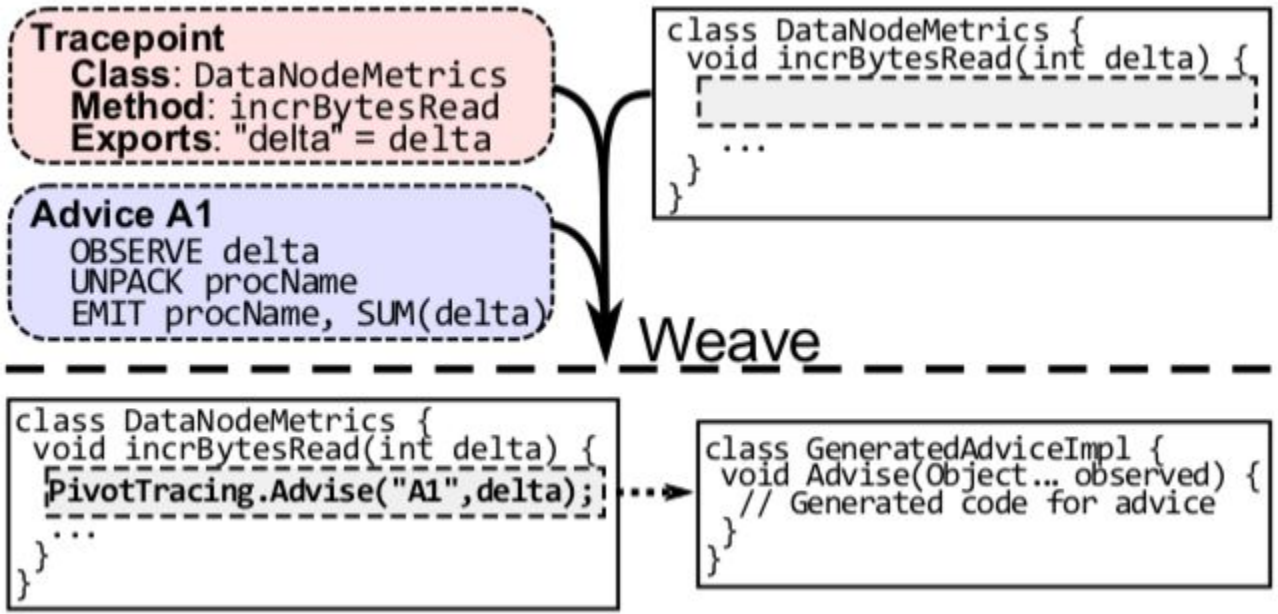
A1 observes and packs procName.
A2 observes delta, unpacks procName and emits (procName, SUM(delta))

Weaving

The process of adding the advice to a tracepoint is called *weaving*.

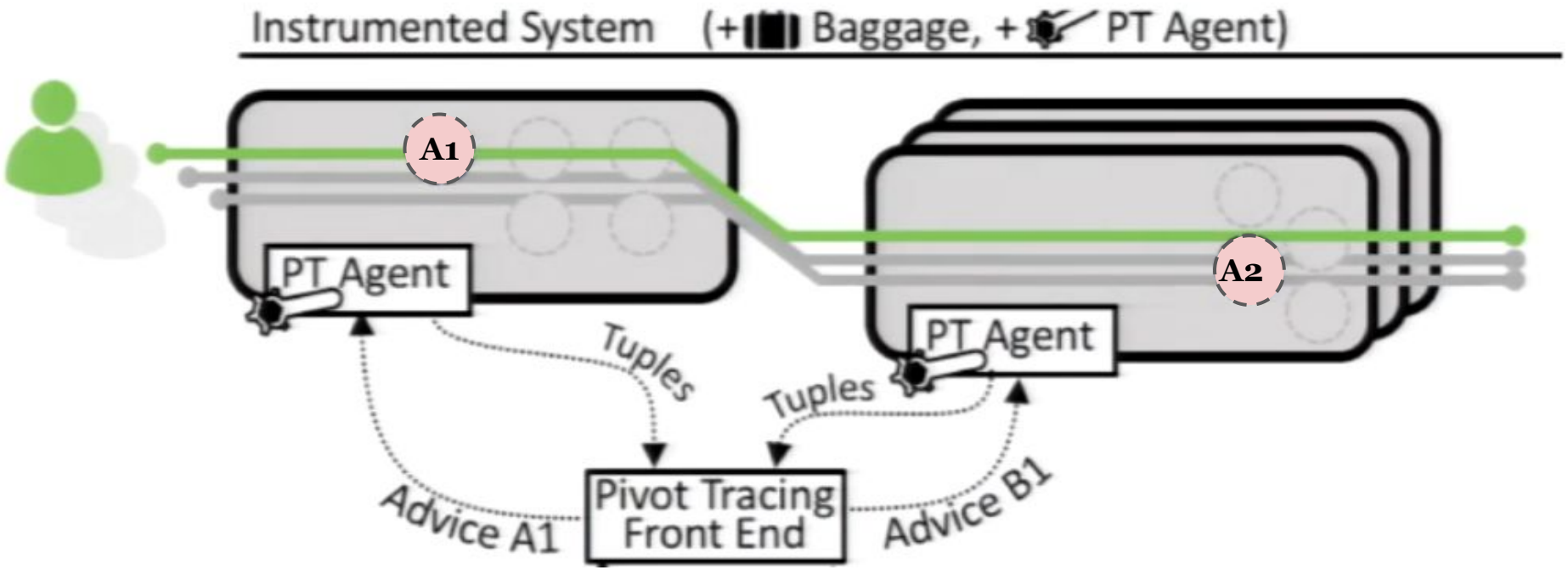
Pivot Tracing contacts the PT Agents responsible for all tracepoints used in the query. It tells them to dynamically load the code that implements the advice operations. The agent inserts the appropriate code at the tracepoint.

Weaving



Variables exported by the tracepoint are passed to the Advise function.

Advice execution example 2



Design and implementation

Implementation

PT Agent

PT agent is a thread that runs in every process in the system.
It uses Javassist to insert code to classes at runtime.
It uses a simple publish-subscribe mechanism to receive commands from PT frontent and to return it emitted tuples.

AspectJ can be used to add `PivotTracing.initialize();` to each main() method in the code.

Baggage

Baggage is a Java library written by the authors. It provides all necessary operation on Baggage objects.
Pivot Tracing requires the code of the system to be modified so that the baggage is passed with each communication between threads, processes, through network or RPC.

Baggage uses Protocol Buffers for serialization, so it can be used between different programming languages.

The library stores the baggage object in thread local storage.

Evaluation

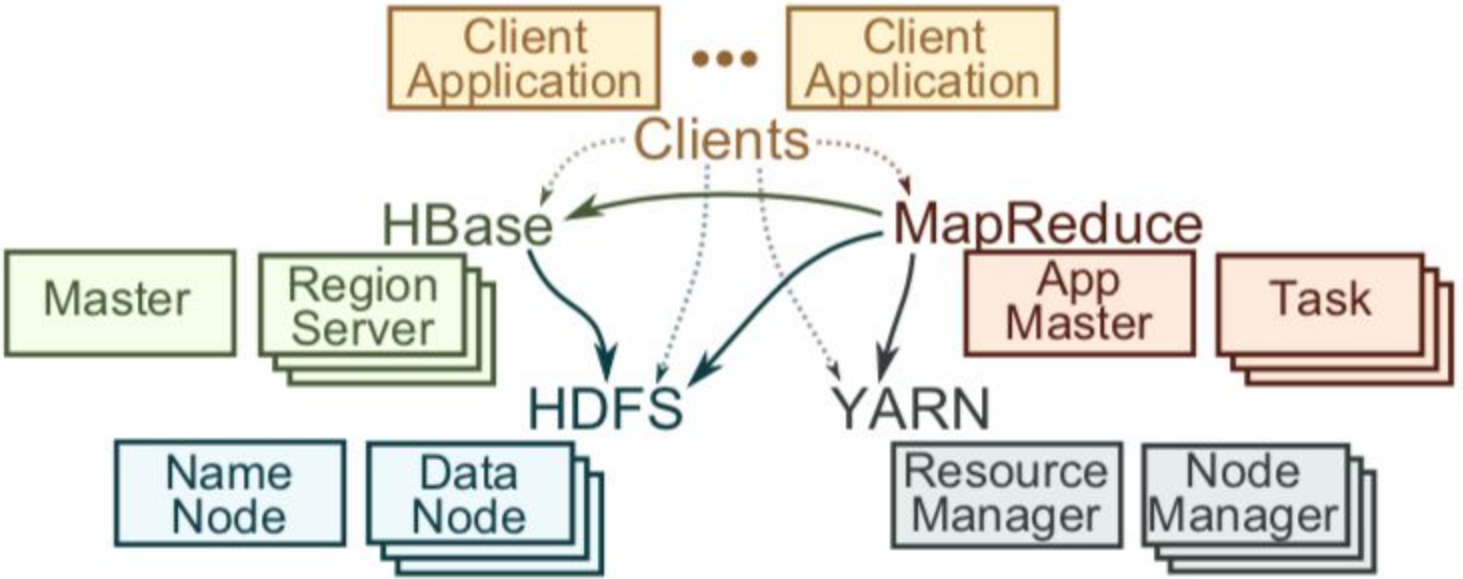
Evaluation

The evaluation was performed on a Hadoop stack. The source code of HDFS, HBase, Hadoop MapReduce and YARN was modified to support Baggage and PT Agents.

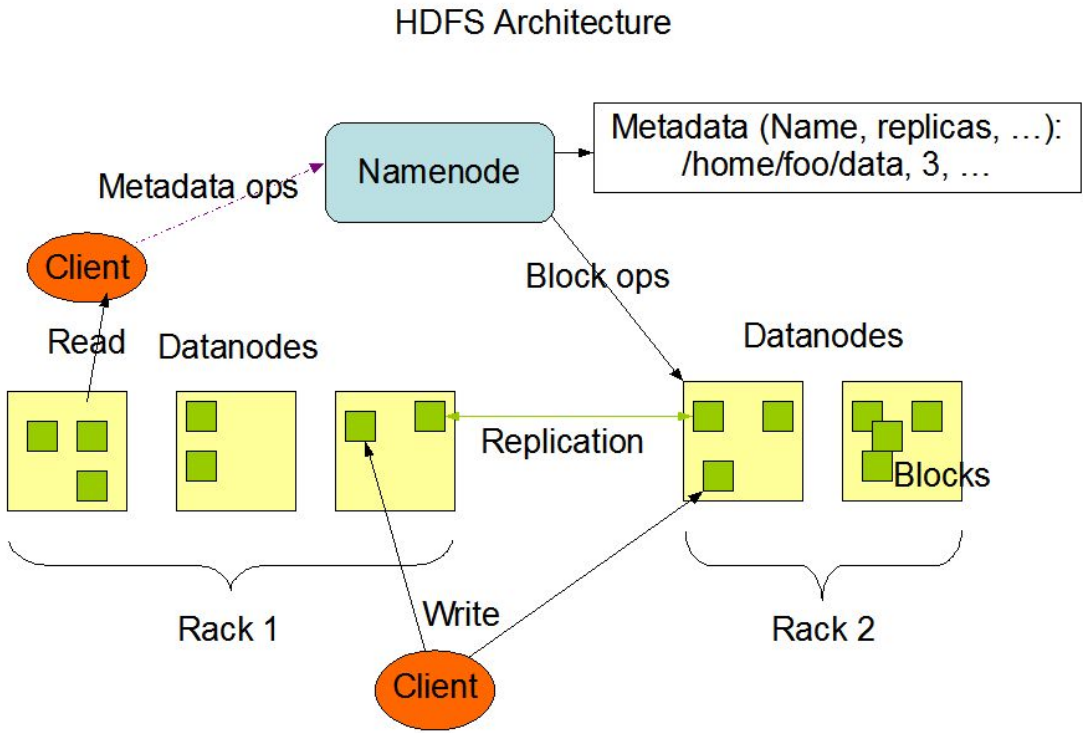
Each system required 50-200 lines of code modifications, mostly in RPC invocations and Threads, Runnables, Callables and Queues.

This was the only required modification to the system.

Evaluation


Client Application
Client Application
Clients
HBase
MapReduce
Master
Region Server
App Master
Task
HDFS
YARN
Name Node
Data Node
Resource Manager
Node Manager

Case study - HDFS Replica Selection Bug



Source: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Case study - HDFS Replica Selection Bug

The client receives nodes that store replicas of a given file. Then he selects one of the replicas:

1) selects local replica
2) selects a rack-local replica
3) selects a random replica

Case study - HDFS Replica Selection Bug

The setup:

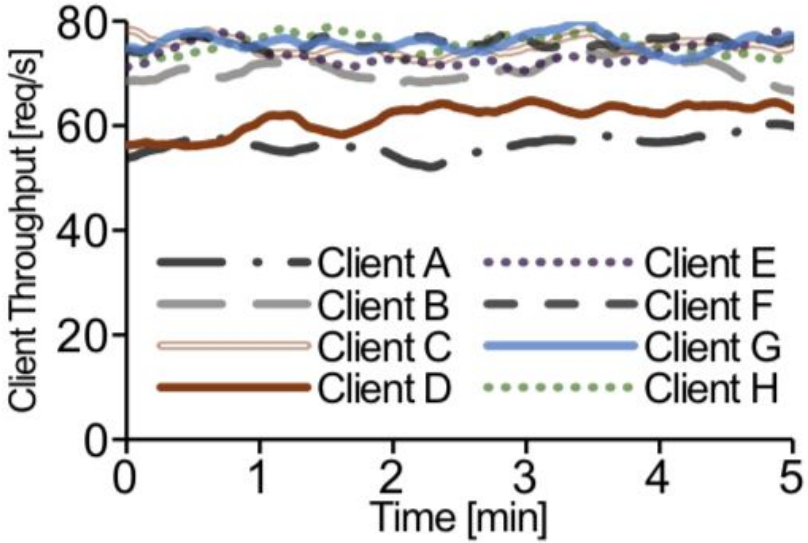
- 96 stress test clients - random 8kB reads for a dataset of 10,000 128MB files
- 1 NameNode
- 8 DataNodes
- each machine has identical hardware specification: 8 cores, 16GB RAM, 1Gbit network interface

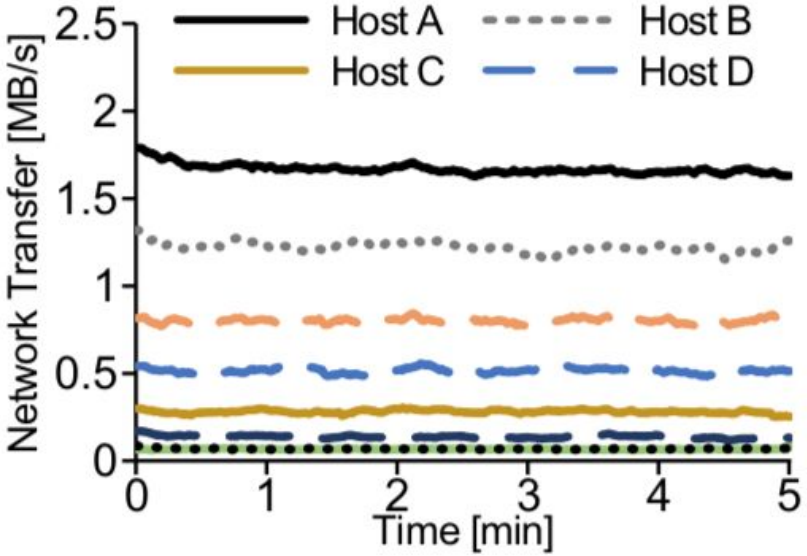
Expectations:

1) uniform throughput from test clients
2) uniform throughput on DataNodes

Case study - HDFS Replica Selection Bug

What actually happened:





Diagnosis with pivot tracing

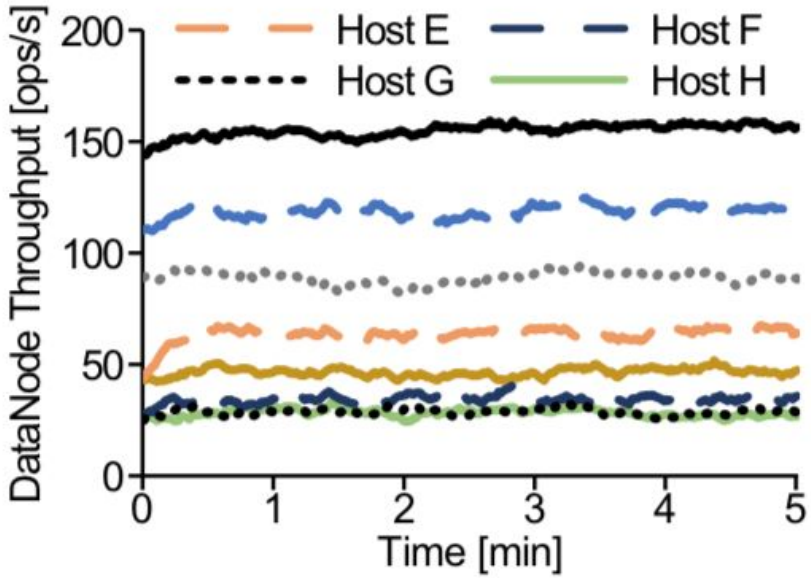
The following query was installed on each DataNode

```
From dnop In DN.DataTransferProtocol
    GroupBy dnop.host
    Select dnop.host, COUNT
```

Diagnosis with pivot tracing

And produced the following results

```
From dnop In DN.DataTransferProtocol
     GroupBy dnop.host
     Select dnop.host, COUNT
```



Diagnosis with pivot tracing

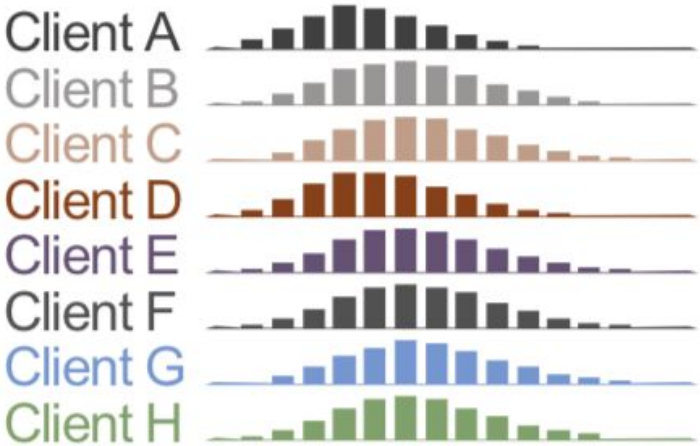
Then the following query was installed on each client and on the NameNode in order to correlate each request with the client that issued it. It counts the number of times each client reads each file.

```
From getloc In NN.GetBlockLocations
    Join st In client.DoNextOp On st -> getloc
    GroupBy st.host, getloc.src
    Select st.host, getloc.src, COUNT
```

Diagnosis with pivot tracing

And produced the results.

```
From getloc In NN.GetBlockLocations
     Join st In client.DoNextOp On st -> getloc
     GroupBy st.host, getloc.src
     Select st.host, getloc.src, COUNT
```



Diagnosis with pivot tracing

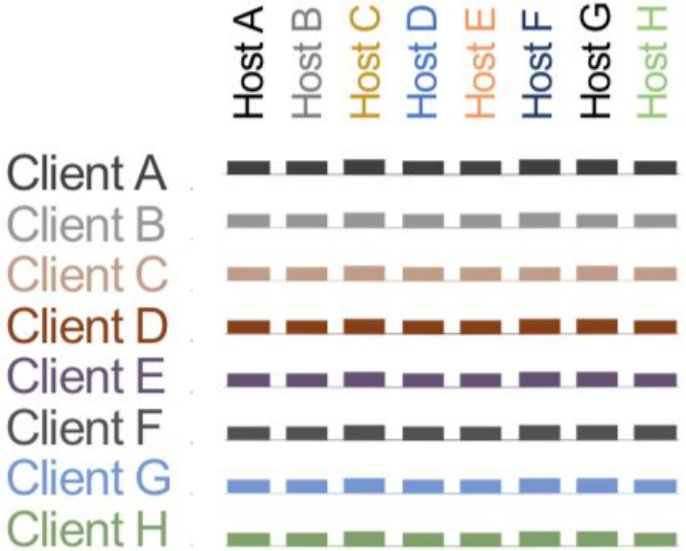
Then the following query was installed on NameNode and each DataNode. It calculates the frequency that each DataNode is hosting a replica for files that are being read.

```
From getloc In NN.GetBlockLocations
    Join st In client.DoNextOp On st -> getloc
    GroupBy st.host, getloc.replicas
    Select st.host, getloc.replicas, COUNT
```

Diagnosis with pivot tracing

Then the following query was installed on NameNode and each DataNode. It calculates the frequency that each DataNode is hosting a replica for files that are being read.

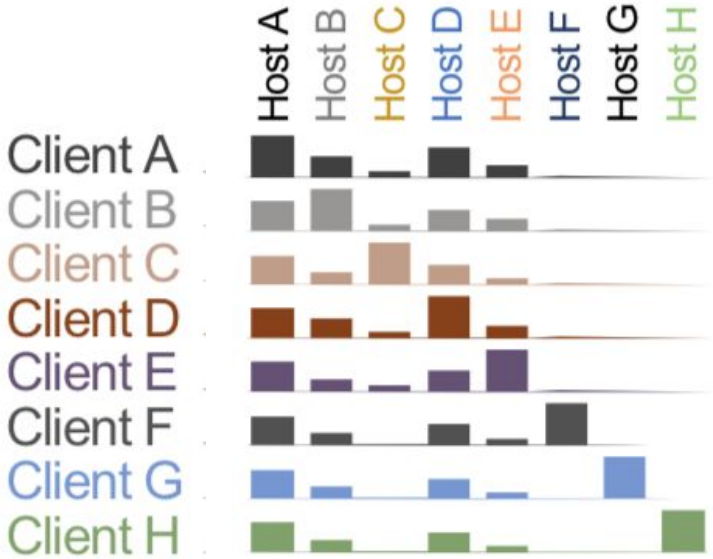
```
From getloc In NN.GetBlockLocations
    Join st In client.DoNextOp On st -> getloc
    GroupBy st.host, getloc.replicas
    Select st.host, getloc.replicas, COUNT
```



Diagnosis with pivot tracing

In order to check the inconsistency, the following query was installed on DataNodes and clients:

```
From dnop In DN.DataTransferProtocol
    Join st In client.DoNextOp On st -> dnop
    GroupBy st.host, dnop.host
    Select st.host, dnop.host, COUNT
```

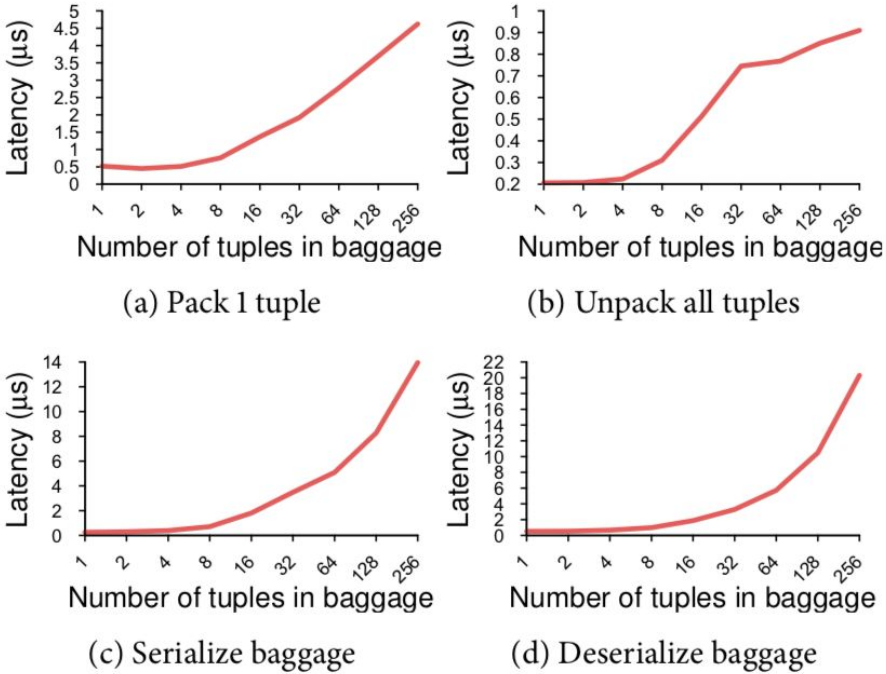


Overhead

Baggage

- Baggage serialization, deserialization, packing and unpacking incurs an overhead.
- By default Pivot Tracing propagates and empty baggage that is serialized to 0 bytes.
- The number of tuples propagated in baggage is reduced by query optimization.
- The size of serialized baggage is approximately linear in the number of packed tuples.

Baggage



Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte tuples.

Application level overhead

Multiple known benchmarks were performed (HiBench, YCSB, HDFS DF-SIO, NNBench). Those benchmark bottleneck on network or disk, but no significant performance change was observed with Pivot Tracing enabled.

To measure the effect of Pivot Tracing on CPU bound requests, the HDFS was stress tested using requests derived from HDFS NNBench:

- `Read8K` reads 8kB from a file
- `Open` opens a file for reading
- `Create` creates a file for writing
- `Rename` renames an existing file

`Read8K` is a DataNode operation, other are NameNode operations.

Application level overhead

	Read8k	Open	Create	Rename
Unmodified	0%	0%	0%	0%
PivotTracing Enabled	0.3%	0.3%	<0.1%	0.2%
Baggage – 1 Tuple	0.8%	0.4%	0.6%	0.8%
Baggage – 60 Tuples	0.82%	15.9%	8.6%	4.1%
Queries – §6.1	1.5%	4.0%	6.0%	0.3%
Queries – §6.2	1.9%	14.3%	8.2%	5.5%

Dynamic instrumentation

JVM HotSwap requires Java's debugging mode to be enabled, which disables some compiler optimizations.
However, the experiments measured only a small overhead between debugging enabled and disabled.
Reloading a class with woven advice has a one-time cost of approximately 100ms, depending on the size of the class.

Conclusion

Conclusion

- this is the first monitoring system that combines dynamic instrumentation and causal tracing
- it allows analysis of systems that consist of multiple applications, operating systems, processes or threads
- the execution overhead is low (?)
- it requires only one-time code modification

This presentation is based on the paper:
http://cs.brown.edu/~rfonseca/pubs/mace15pivot.pdf

And on a presentation presented by one of the authors:
https://www.youtube.com/watch?v=CaRgYTLqJnI