LinkedIn

pinot

Query Processing Resiliency

Strengthening Apache Pinot's Query Processing Engine with Adaptive Server Selection and Runtime Query Killing

Community Over Code ASF Conference 2023

Vivek Iyer Vaidyanathan

Software Engineer,
LinkedIn

Jia Guo

Software Engineer,
LinkedIn

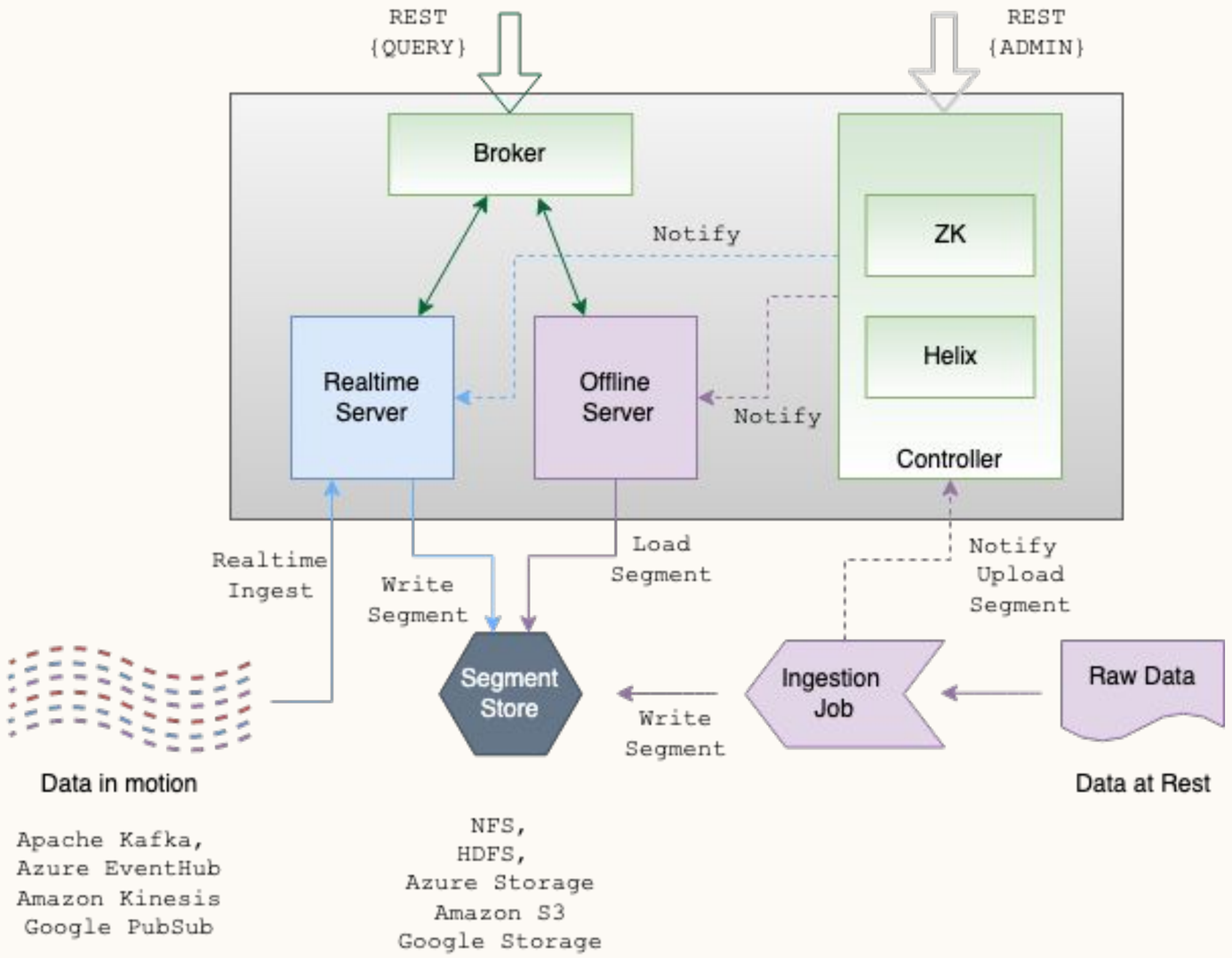
Agenda

1. Pinot at LinkedIn
2. Adaptive Server Selection
3. Runtime Query Killing
4. Future Work and Conclusion

What is Apache Pinot?

- OLAP Datastore
- Lambda architecture
  - Offline data pushes + Real-time stream ingestion
- Low latency analytics
- Columnar, indexed storage
- Distributed – highly available, reliable, scalable

Pinot Architecture



Powered by Apache Pinot







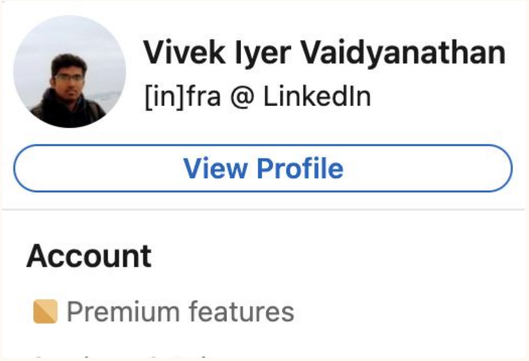


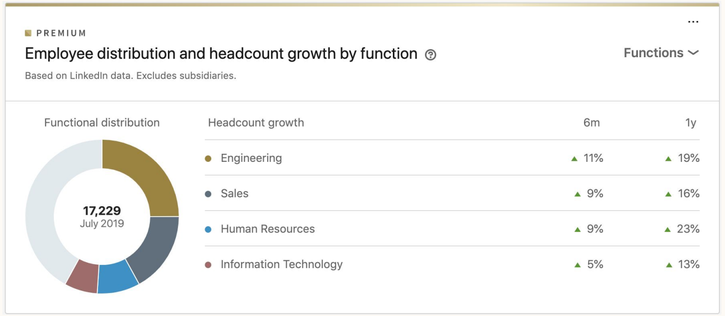


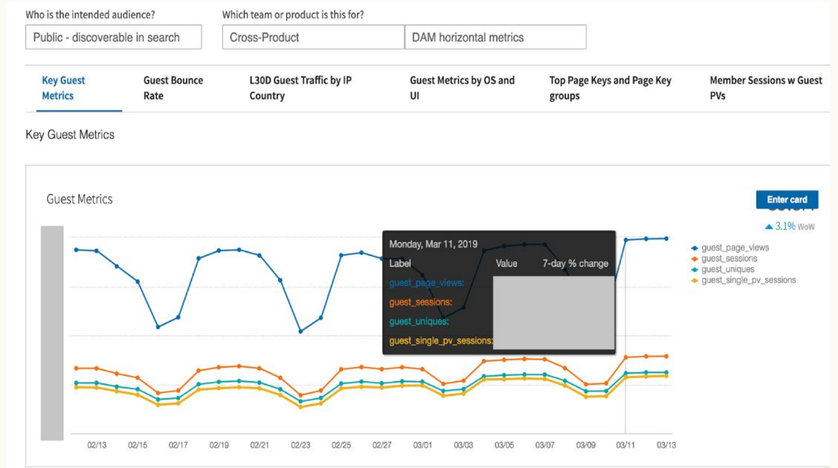


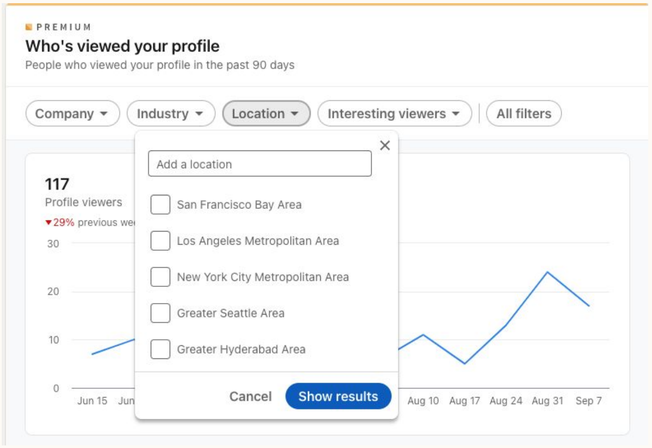


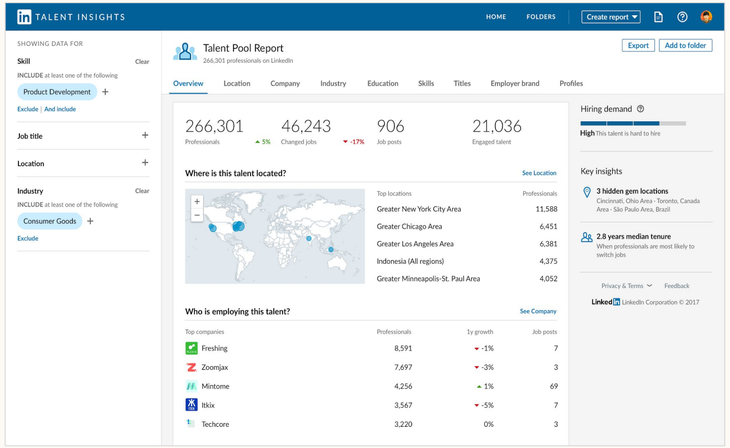

Pinot @ LinkedIn











Member Facing Analytics

Enterprise Analytics Products

Internal Products and BI Tools

```java
/*
 * The Hybrid score for each server is calculated as follows. The server with the lowest Hybrid score is picked.
 *       HybridScore = Math.pow(A+B, N) * C
 * N -> Configurable exponent with default value of 3.
 */
public class HybridSelector implements AdaptiveServerSelector {
  private final ServerRoutingStatsManager _serverRoutingStatsManager;
  private final Random _random;

  public HybridSelector(ServerRoutingStatsManager serverRoutingStatsManager) {
    _serverRoutingStatsManager = serverRoutingStatsManager;
    _random = new Random();
  }

  @Override
  public String select(List<String> serverCandidates) {
    String selectedServer = null;
    Double minScore = Double.MAX_VALUE;

    // TODO: If two or more servers have the same score, break the tie intelligently.
    for (String server : serverCandidates) {
      Double score = _serverRoutingStatsManager.fetchHybridScoreForServer(server);

      // No stats for this server. That means this server hasn't received any queries yet.
      if (score == null) {
        int randIdx = _random.nextInt(serverCandidates.size());
        selectedServer = serverCandidates.get(randIdx);
        break;
      }

      if (score < minScore) {
        minScore = score;
        selectedServer = server;
      }
    }

    return selectedServer;
  }

  @Override
  public List<Pair<String, Double>> fetchAllServerRankingsWithScores() {
    List<Pair<String, Double>> pairList = _serverRoutingStatsManager.fetchHybridScoreForAllServers();

    // Let's shuffle the list before sorting. This helps with randomly choosing different servers if there is a tie.
    Collections.shuffle(pairList);
    Collections.sort(pairList, (o1, o2) -> {
      int val = Double.compare(o1.getRight(), o2.getRight());
      return val;
    });

    return pairList;
  }
```

Adaptive Server Selection

LinkedIn

Adaptive Server Selection

Agenda

1. **Problem Statement**
2. Design
3. Regression Benchmarking
4. Results after prod rollout at LinkedIn

Problem Statement

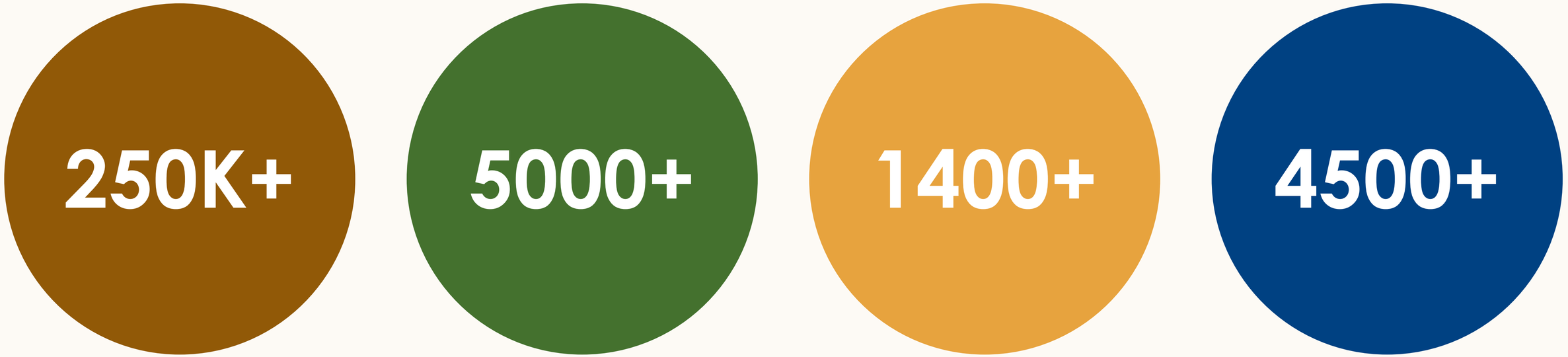
Query Routing in Pinot

- Pinot Servers host the segments that contain the data to be queried
- Each segment is hosted on multiple servers controlled by replication factor
- Pinot Broker receives the query
- Broker uses a round-robin approach to pick the servers to process a query

Issues with Round-robin Routing

- Pinot Servers are susceptible to both transient and permanent slowness issues - GC Pause, network issues, and disk failures
- With round-robin selection, queries are sent to servers regardless of server performance, which can result in slower/failed responses.
- A more intelligent approach is needed to optimize server selection and improve overall performance.

Scale @ LinkedIn



Pain Points @ Linkedin Scale

- Pinot team spends ~72+ engineering hours every quarter to troubleshoot server slowness issues
- Customers face availability degradation when there are Pinot server failures
- Breaches to Latency SLAs when Pinot Servers experience slowness or failures - 30+ events per quarter
- Elevated urgency when triaging latency increase alerts

Adaptive Server Selection

Agenda

1. Problem Statement
2. **Design**
3. Regression Benchmarking
4. Results after prod rollout at LinkedIn

Design Components

Building Blocks of the feature

Stats Collector

- Local to each broker
- Stats are stored in-memory
- Per-server stats that are maintained

  # of in-inflight queries

  EWMA of in-flight queries

  EWMA of latencies observed

+

Server Selector

- Uses an intelligent selection policy to pick the best server
- Decisions are made based on local state
- Selection policies supported

  Uses the # of in-inflight queries

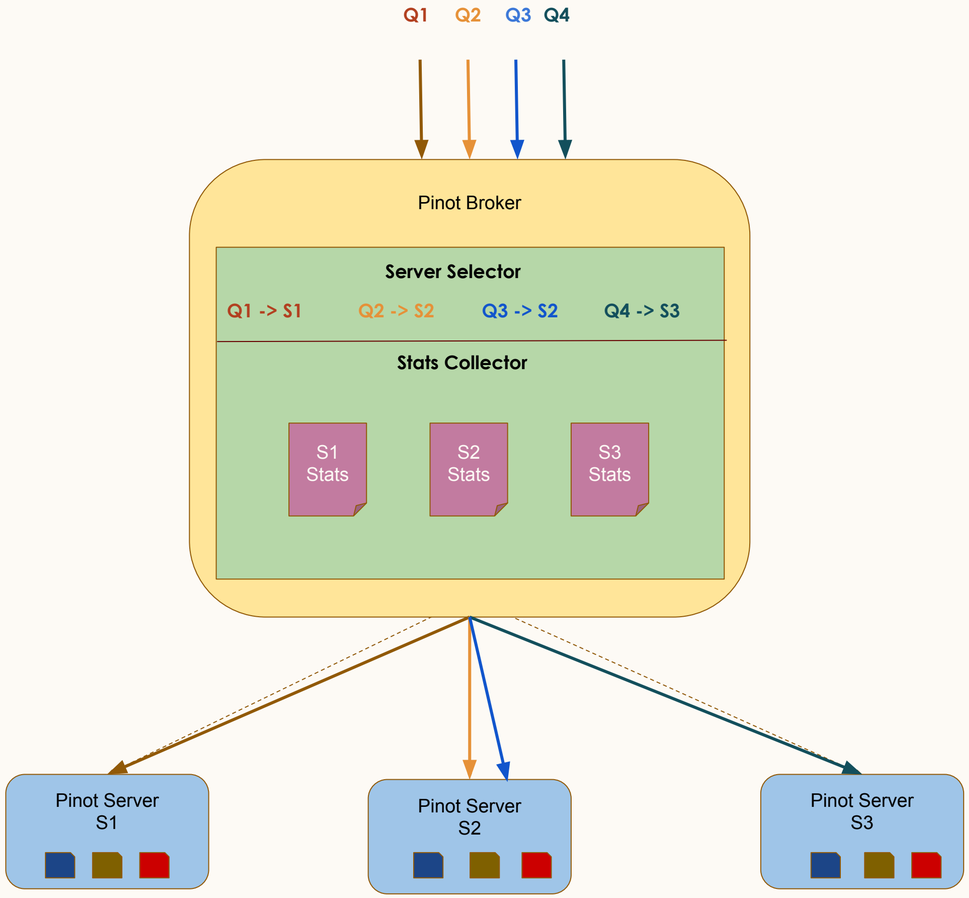
  Uses latencies

  Sophisticated policy using latency and # of in-flight queries

=

Smarter Query Routing at Broker

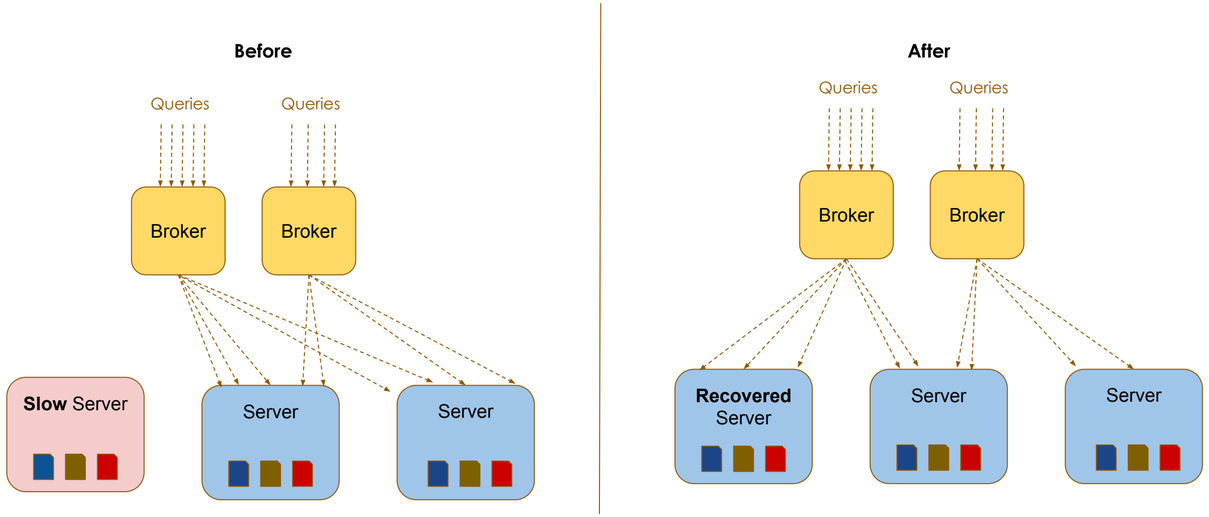
Workflow



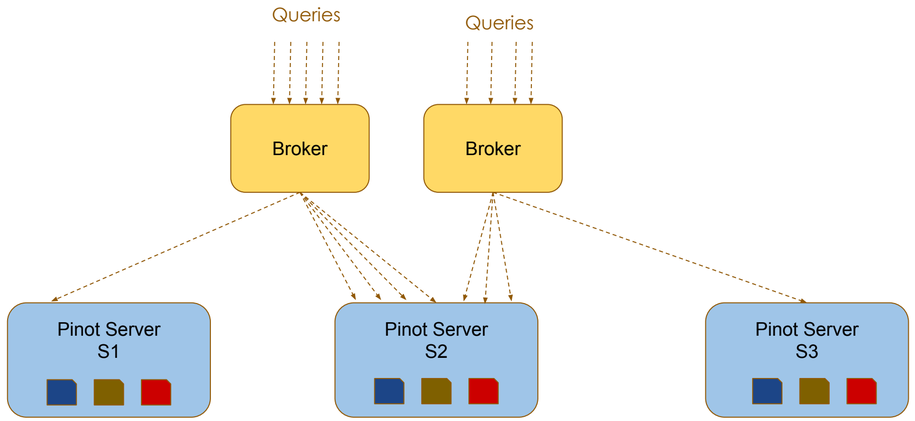
Stats Collector

- Async stats collection for minimal overhead
- Before routing to server, # of in-flight requests are updated
- After receiving response from server, latency and # of inflight requests are updated

Auto Recovery



Herd Behavior



Server Selection Policy

- Minimal overhead to query processing
- Quick Detection: Must quickly detect slow servers and tune down QPS.
- Auto Recovery: Must cope and quickly react when servers recover
- Well-behaved: Must avoid herd behaviors
- Used independently, signals like current Q size and Latency are raw and delayed

Hybrid Server Selector

- Rank each server based on a score. Pick the server with the lowest score.

$$\text{ServerScore} = (\text{estimated\_queue})^{N} \times \text{Latency}_{EWMA}$$

$$\text{estimated\_queue} = Q + Q_{EWMA} + 1$$

- EWMA smoothens the values giving priority to changes in recent past
- Avoids herd behavior by forecasting the future Q size size for a server
- Reacts faster to changes on the server by using an exponential function

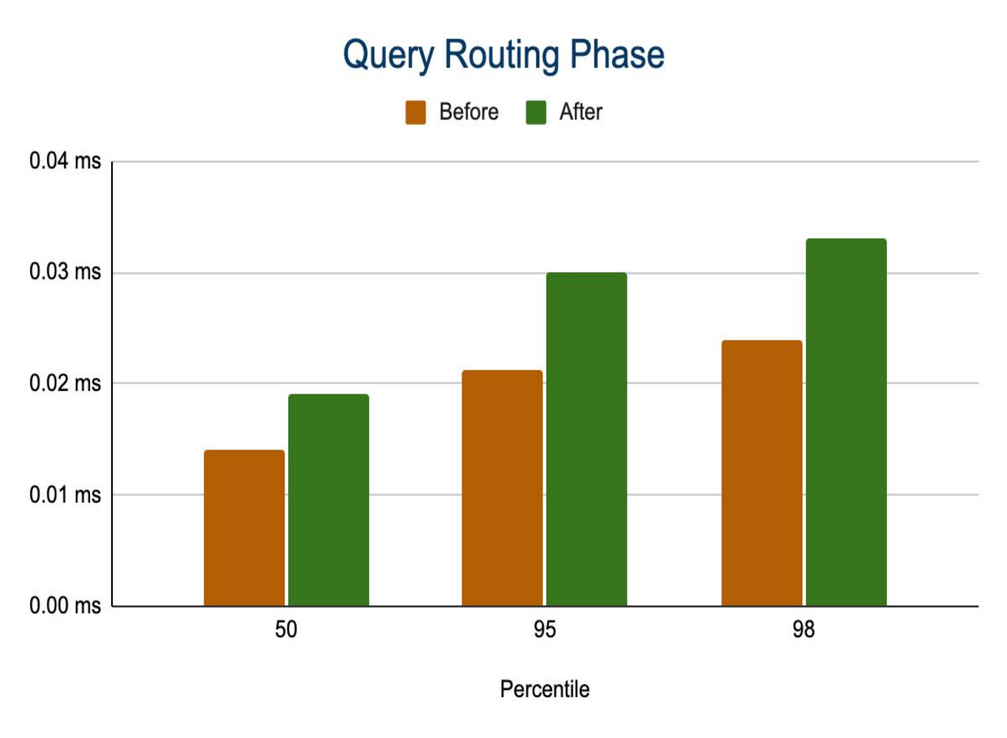
Adaptive Server Selection

Agenda

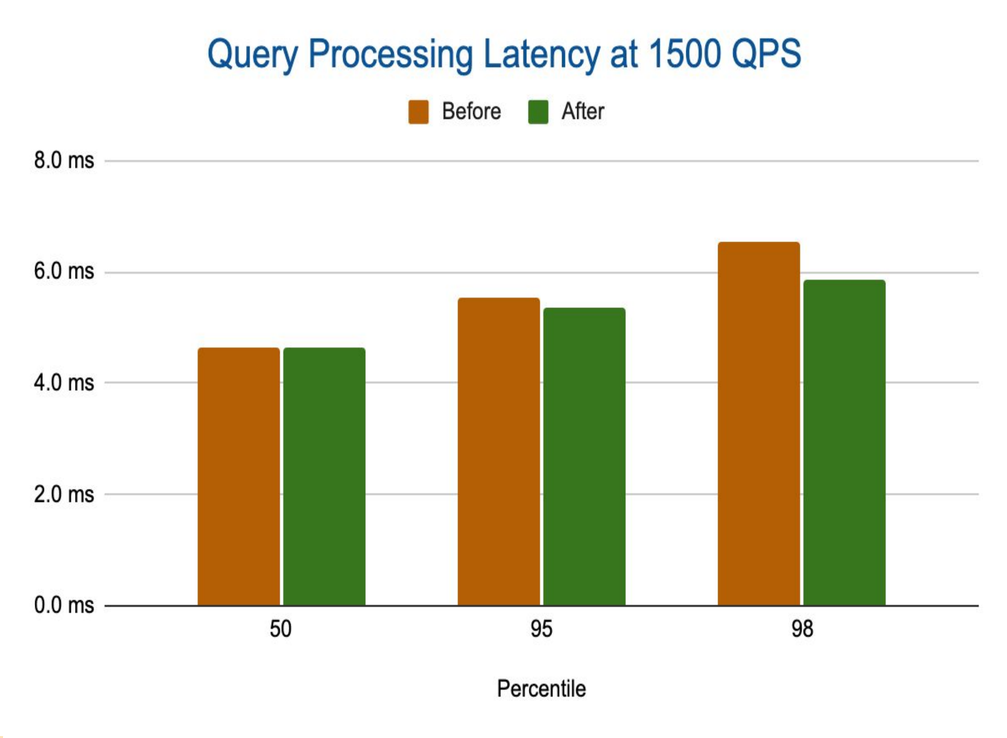
1. Problem Statement
2. Design
3. **Regression Benchmarking**
4. Results after prod rollout at LinkedIn

Latency Overhead

Query Routing Phase



Total Query Processing

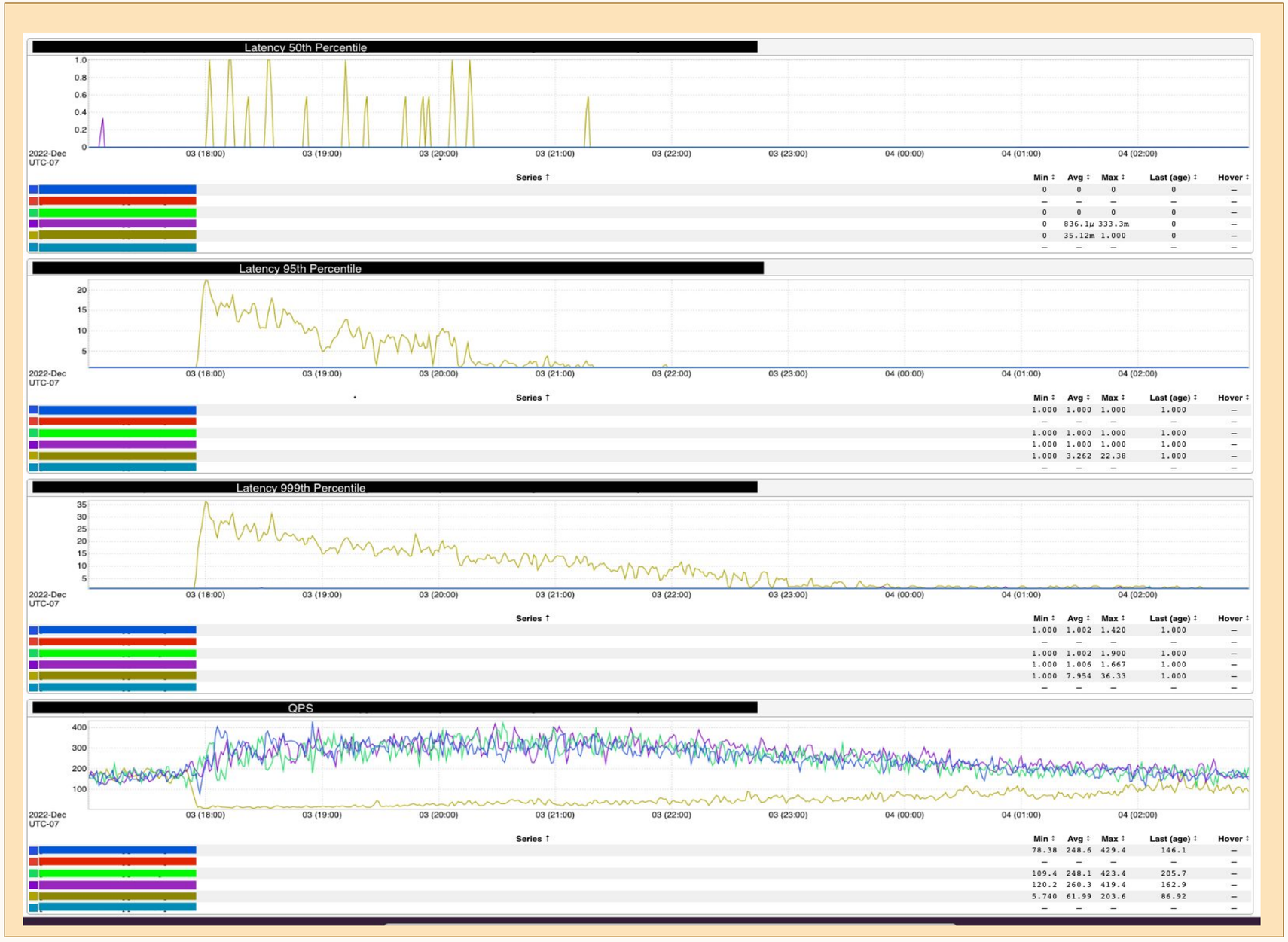


Adaptive Server Selection

Agenda

1. Problem Statement
2. Design
3. Regression Benchmarking
4. **Results after prod rollout at LinkedIn**

Prolonged Server Slowness



Transient Server Slowness

Outcome Highlight

60

Before
Number of slow server latency alerts per quarter

2

After
Number of slow server alerts per quarter

Outcome Highlight

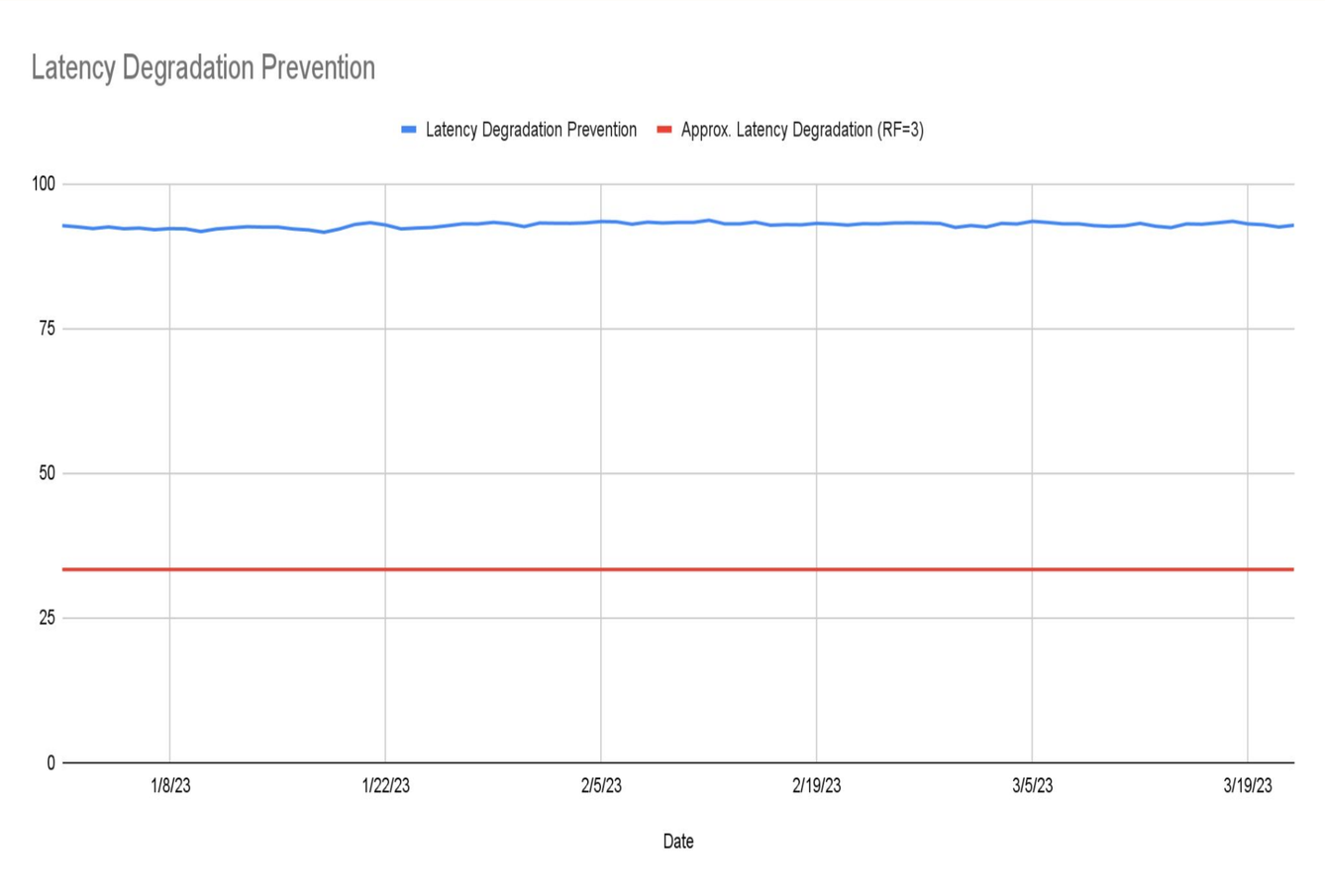
72

Before
Number of engineering hours spent in debugging transient server slowness issues

8

After
Number of engineering hours spent in debugging transient server slowness issues

At **LinkedIn**, this work helped prevent latency degradation in the event of slowness on servers for more than 90% of queries in production.



Prevention of Latency Degradation

- Single server slowness causes latency degradation for ~ 33.33% of queries when RG = 3
- Adaptive Server Selection reduces the chances of latency degradation when one or more servers slow down.

```java
/**
 * Kill the query with the highest cost (memory footprint/cpu time/...)
 * Will trigger gc when killing a consecutive number of queries
 * use XX:+ExplicitGCInvokesConcurrent to avoid a full gc when system.gc is triggered
 */
private void killMostExpensiveQuery() {
  if (!_aggregatedUsagePerActiveQuery.isEmpty() && _numQueriesKilledConsecutively >= _gcTriggerCount) {
    System.gc();
    LOGGER.warn("Triggered gc after killing {} queries", _numQueriesKilledConsecutively);
    _numQueriesKilledConsecutively = 0;
    try {
      Thread.sleep(_normalSleepTime);
    } catch (InterruptedException ignored) {
    }
    _usedBytes = MEMORY_MX_BEAN.getHeapMemoryUsage().getUsed();
    if (_usedBytes < _criticalLevel) {
      return;
    }
  }
  if (!(_isThreadMemorySamplingEnabled || _isThreadCPUSamplingEnabled)) {
    LOGGER.warn("Heap used bytes {} exceeds critical level", _usedBytes);
    LOGGER.warn("But unable to kill query because neither memory nor cpu tracking is enabled");
    return;
  }
  // Critical heap memory usage while no queries running
  if (_aggregatedUsagePerActiveQuery.isEmpty()) {
    LOGGER.debug("Heap used bytes {} exceeds critical level, but no active queries", _usedBytes);
    return;
  }
  AggregatedStats maxUsageTuple;
  if (_isThreadMemorySamplingEnabled) {
    maxUsageTuple = Collections.max(_aggregatedUsagePerActiveQuery.values(),
        Comparator.comparing(AggregatedStats::getAllocatedBytes));
    boolean shouldKill = _oomKillQueryEnabled && maxUsageTuple._allocatedBytes > _minMemoryFootprintForKi
    if (shouldKill) {
      maxUsageTuple._exceptionAtomicReference
          .set(new RuntimeException(String.format(
              " Query %s got killed because using %d bytes of memory on %s, exceeding the quota",
              maxUsageTuple._queryId, maxUsageTuple.getAllocatedBytes(), _instanceType)));
      interruptRunnerThread(maxUsageTuple.getAnchorThread());
    }
    LOGGER.error("Heap used bytes {} exceeds critical level {}", _usedBytes, _criticalLevel);
    LOGGER.error("Query {} got picked because using {} bytes of memory, actual kill committed {}",
        maxUsageTuple._queryId, maxUsageTuple._allocatedBytes, shouldKill);
  } else {
```

OOM Protection Using Automatic Query Killing

LinkedIn

Query Killing

Agenda

1. **Motivation and Challenges**
2. Design
3. Result after prod rollout at LinkedIn

Pain Points @ LinkedIn

- CPU/memory intensive query can silently slow down other queries
- Server crashes leading to SLAs breaches & availability degradation
- The user gets no proper warnings for expensive queries
- Difficulty triaging OOM exceptions and identifying expensive queries

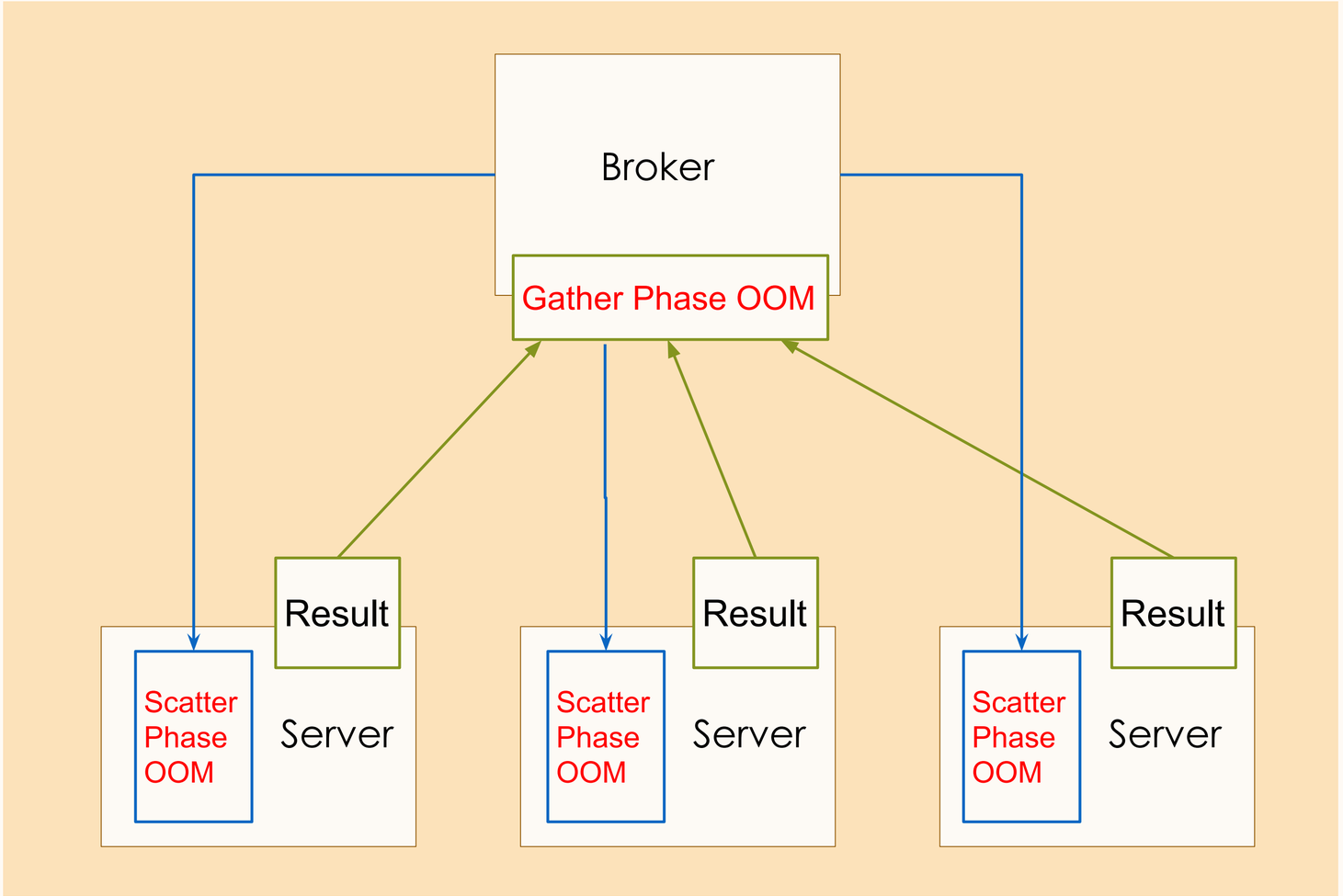
Problem Statement

Goals

- OOM protection for servers and brokers
- Kill high-risk queries on the fly

Challenges

- No runtime memory tracking for Pinot queries
- Pinot's multi-threaded query execution model using threadpools
- Java's opaque memory management
- Overhead of memory accounting



Query Killing

Agenda

1. Motivation and Challenges
2. **Design:**
   - Stats collection
   - Stats Aggregation / Accounting
3. Results after prod rollout at LinkedIn

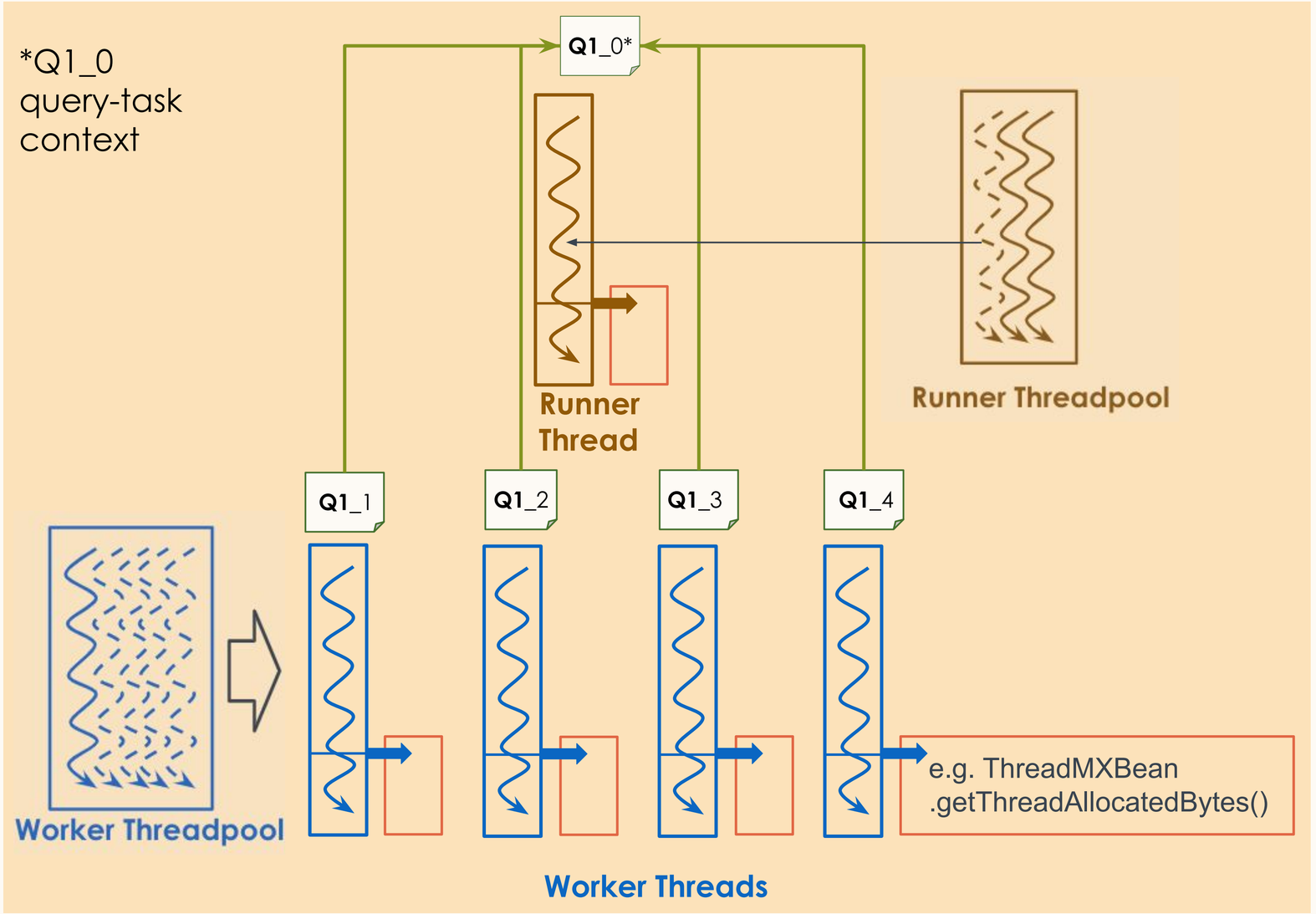
Stats Collection - Overview

Two parts to Stats Collection:

- Setting up query-task context
- Instrumentation

Characteristics:

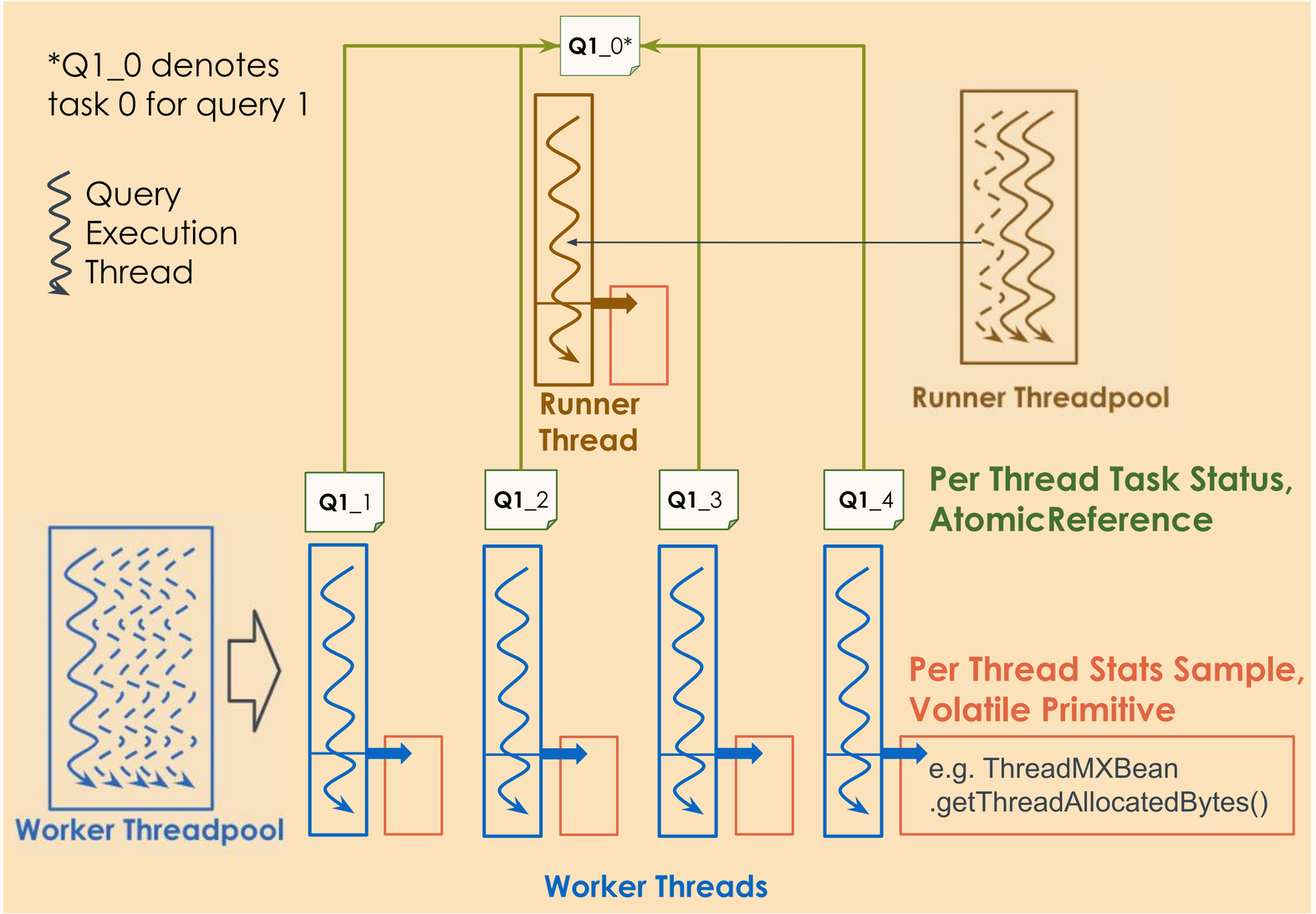
- Generic tree-like context model
- Thread-reported generic usages
- Lock Free: low overhead



Getting Usage Statistics with Instrumentation

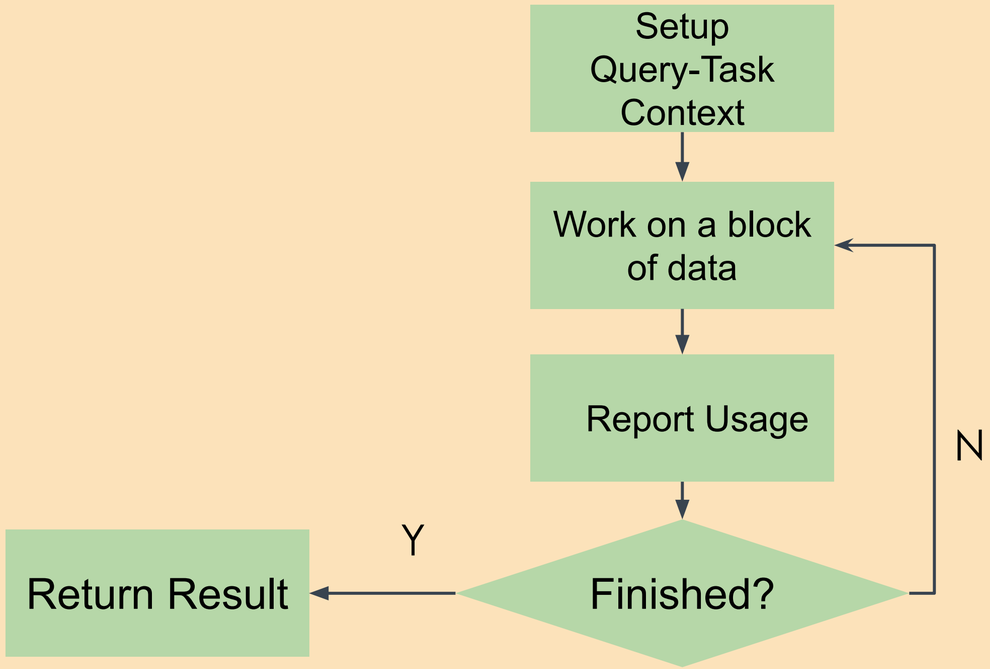
Generic Status/Usage Reporting for Multi-threaded Query Execution Code

- Tree-like runtime query status context
- Thread-reported generic usages
- Lock Free : low overhead



Stats Collection - Example

Query Execution Thread (worker thread as an example)



Stats Collection: Example

Setting Up Query-Task Context

Runner Thread

Worker Threads

```
protected void startProcess() {
  Tracing.activeRecording().setNumTasks(_numTasks);
  ThreadExecutionContext parentContext = Tracing.getThreadAccountant().getThreadExecutionContext();
  for (int i = 0; i < _numTasks; i++) {
    int taskId = i;

    _futures[i] = _executorService.submit(new TraceRunnable() {
      @Override
      public void runJob() {
        ThreadResourceUsageProvider threadResourceUsageProvider = new ThreadResourceUsageProvider();

        Tracing.ThreadAccountantOps.setupWorker(taskId, threadResourceUsageProvider, parentContext);

        // ...

        try {
          processSegments();
        } catch (EarlyTerminationException e) {
          // Early-terminated by interruption (canceled by the main thread)
        } catch (Throwable t) {
          // ...
        } finally {
          onProcessSegmentsFinish();
          _phaser.arriveAndDeregister();
          Tracing.ThreadAccountantOps.clear();
        }

        _totalWorkerThreadCpuTimeNs.getAndAdd(threadResourceUsageProvider.getThreadTimeNs());
      }
    });

  }
}
```

Get Context from the Runner Thread

Create Worker query-thread context

Clear Context When Query Finishes

Stats Collection: Example

Instrumenting code

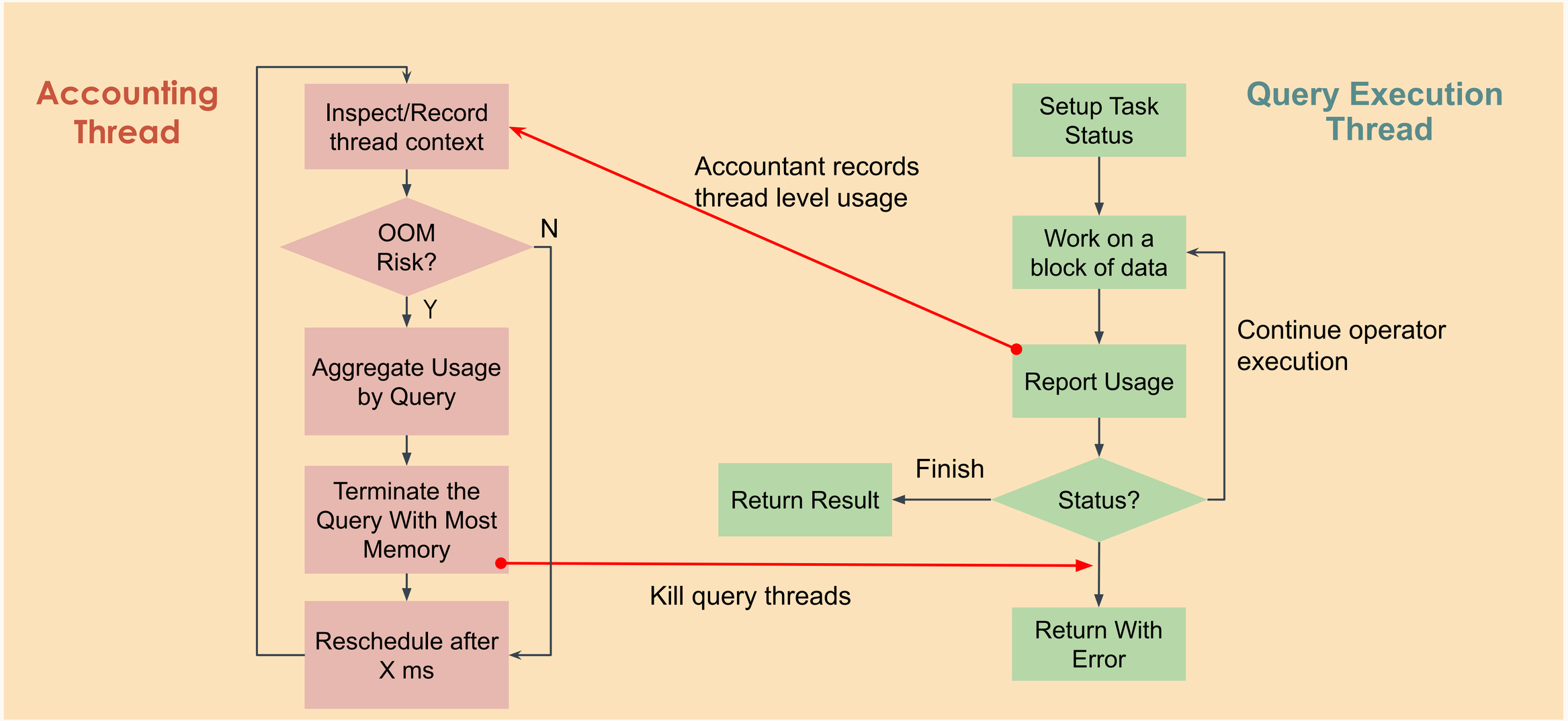
```
1   @Override
2   protected DocIdSetBlock getNextBlock() {
3     if (_currentDocId == Constants.EOF) {
4       return null;
5     }
6
7     // Initialize filter block document Id set
8     if (_blockDocIdSet == null) {
9       _blockDocIdSet = _filterOperator.nextBlock().getBlockDocIdSet();
10      _blockDocIdIterator = _blockDocIdSet.iterator();
11    }
12
13    Tracing.ThreadAccountantOps.sample();
14
15    int pos = 0;
16    int[] docIds = THREAD_LOCAL_DOC_IDS.get();
17
18    // ...
```

One-shot Usage Collection:
Inject 1 line of code
in the operator execution
codepath

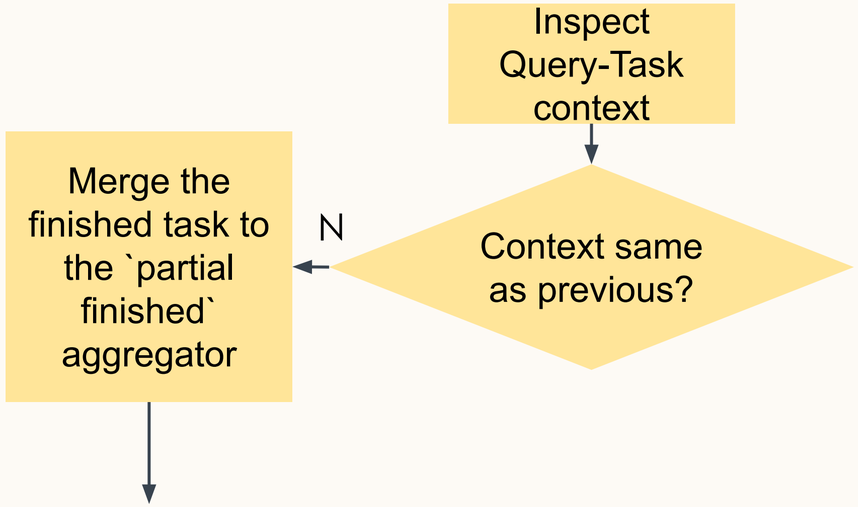
AggregationFunction
GroupByOperator
TransformOperator
ProjectionOperator
DocIdSetOperator
A Block of Data

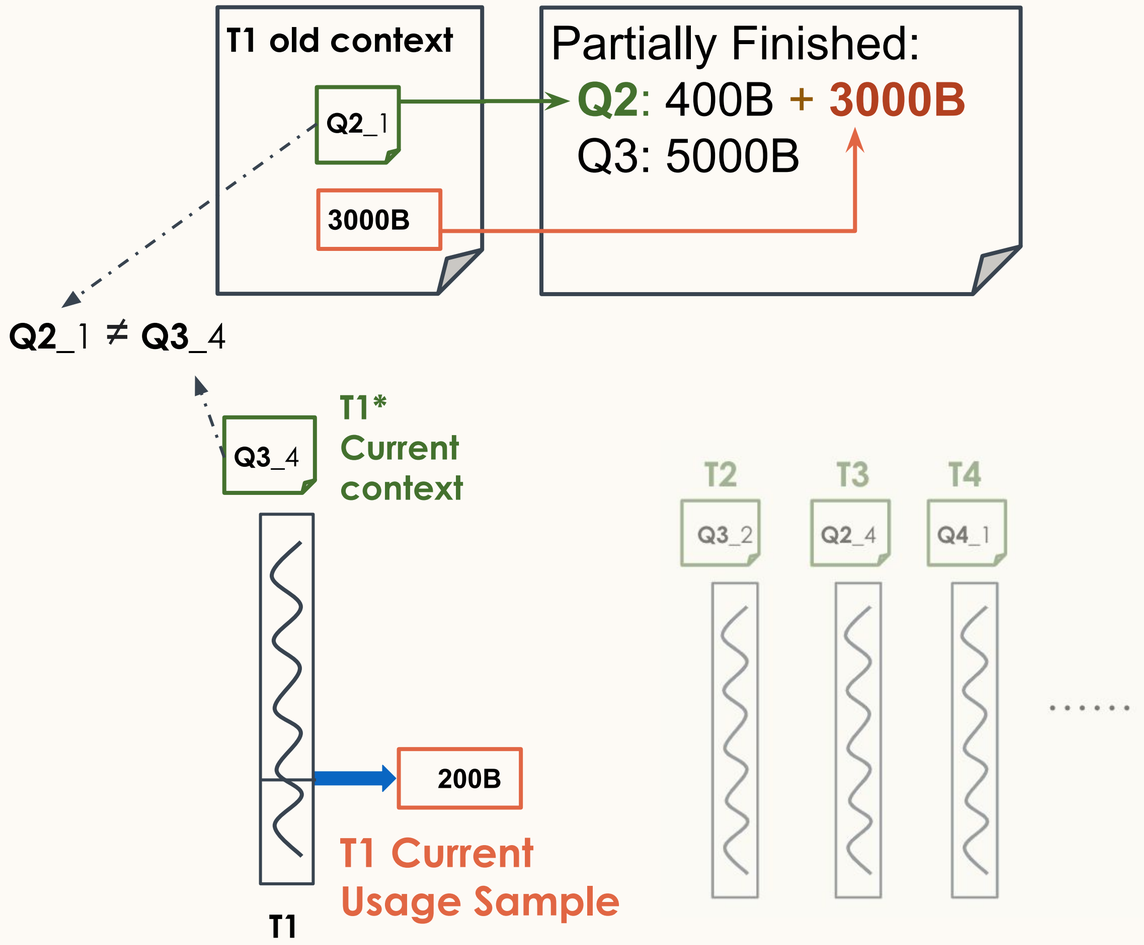
Stats Aggregation/Accounting

Building Accounting/Killing upon Execution Instrumentation



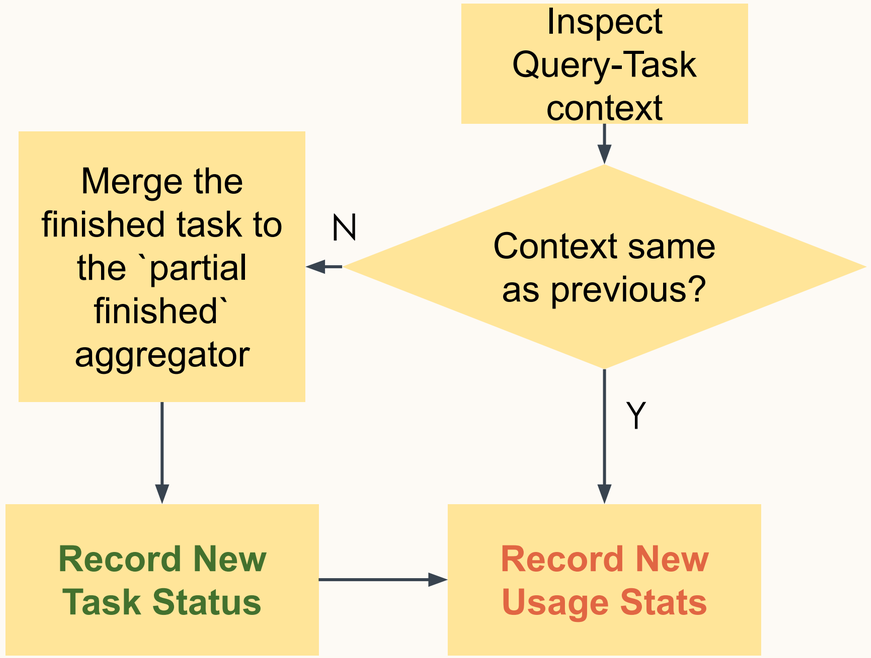
Query Usage Accounting Algorithm

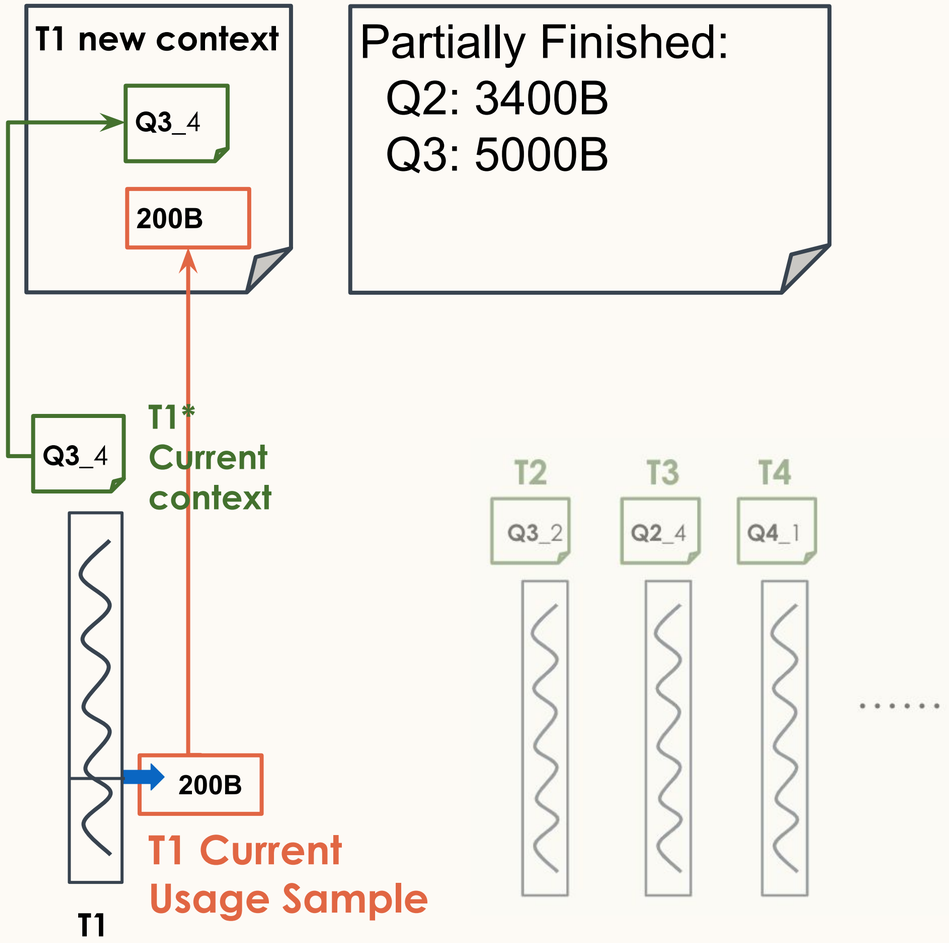




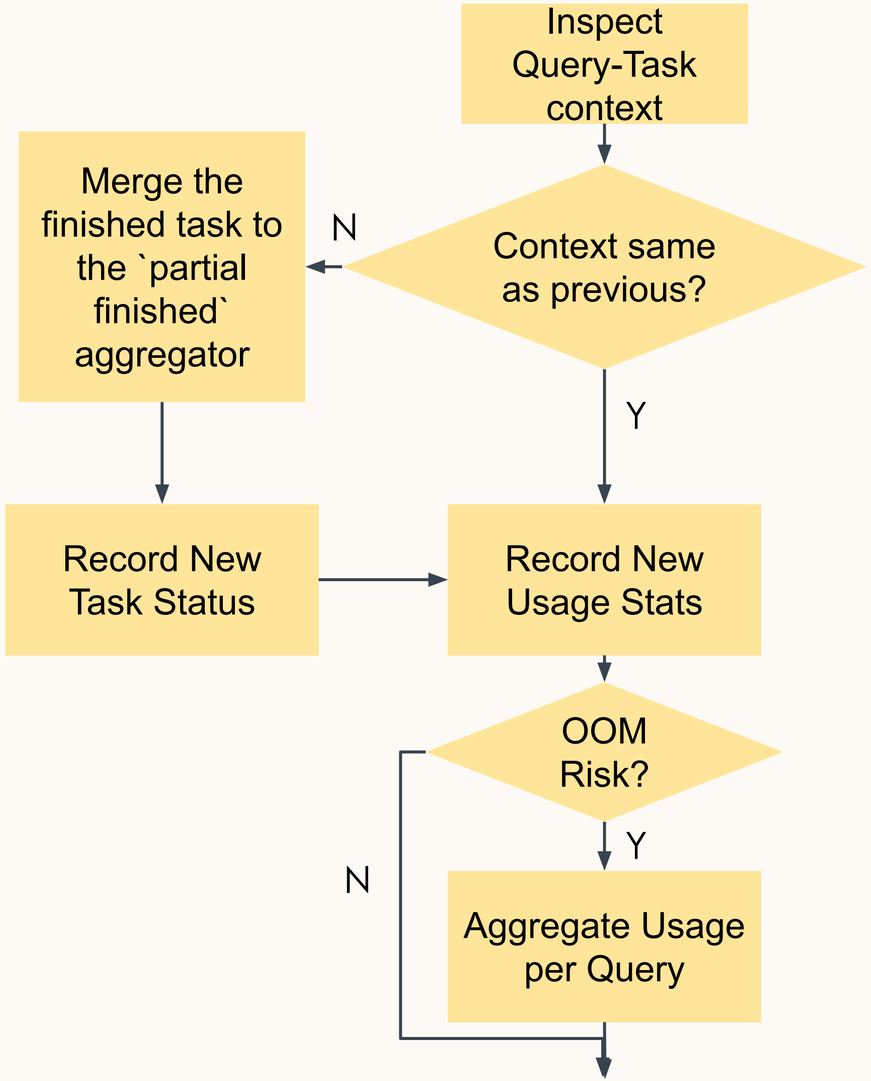
*For simplicity we demonstrate only 1 thread from the threadpool

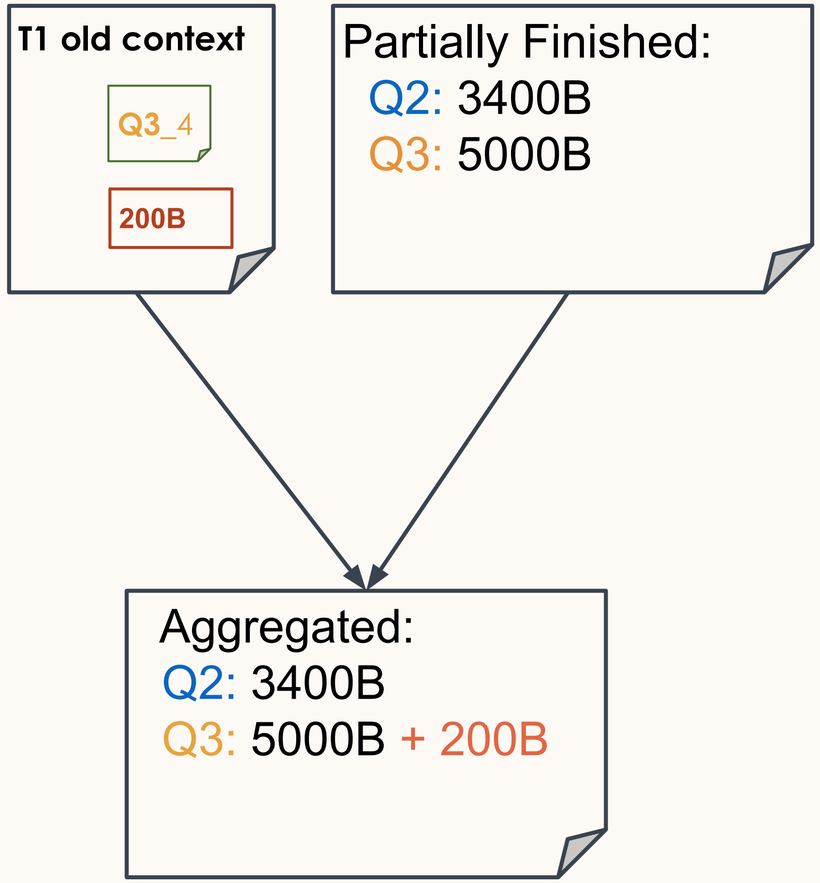
Query Usage Accounting Algorithm



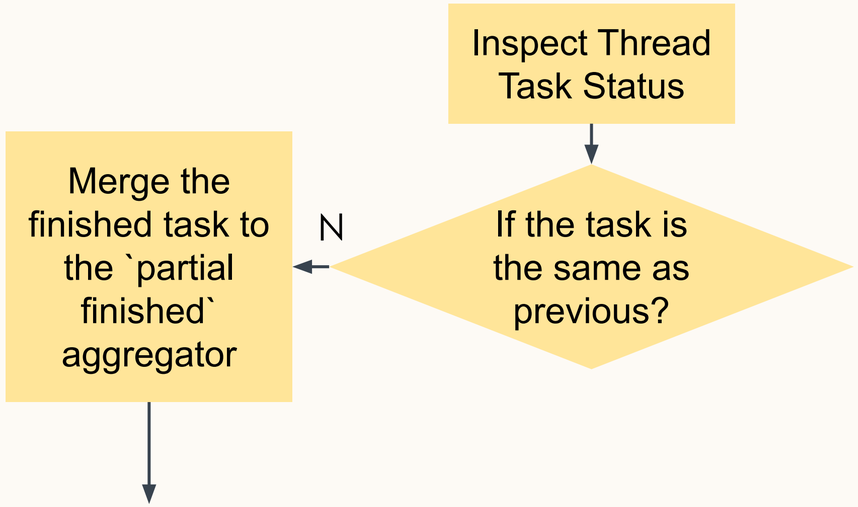


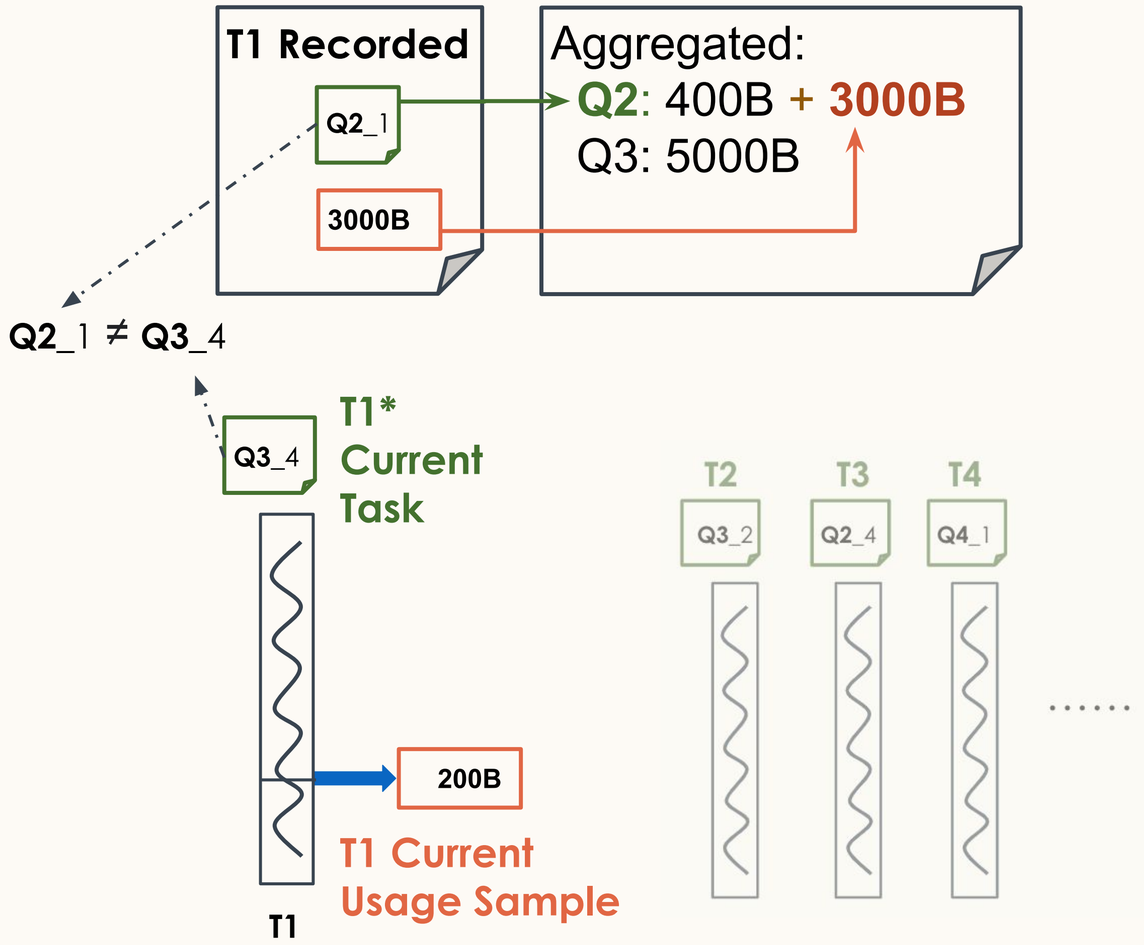
Query Usage Accounting Algorithm





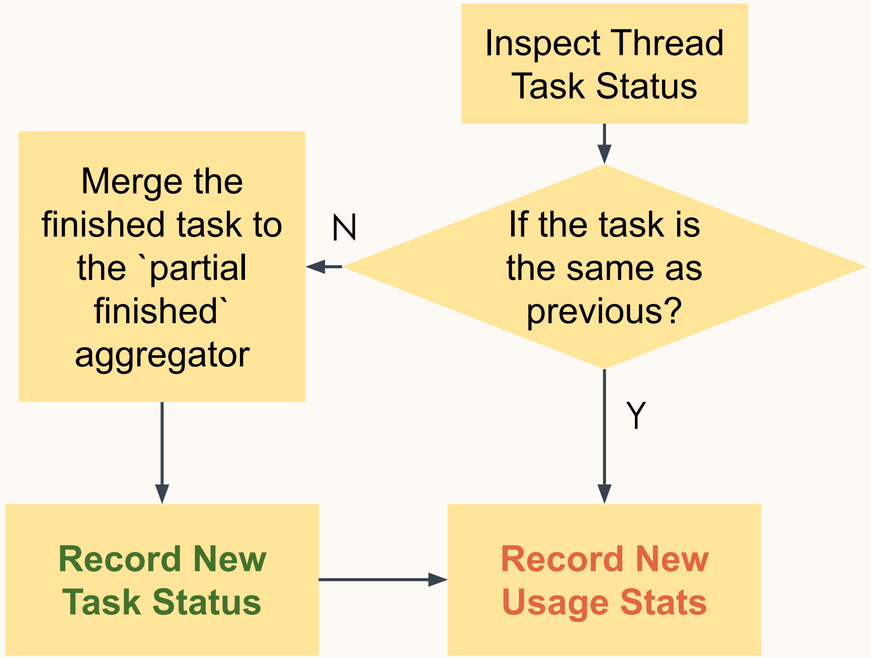
Query Usage Accounting Algorithm

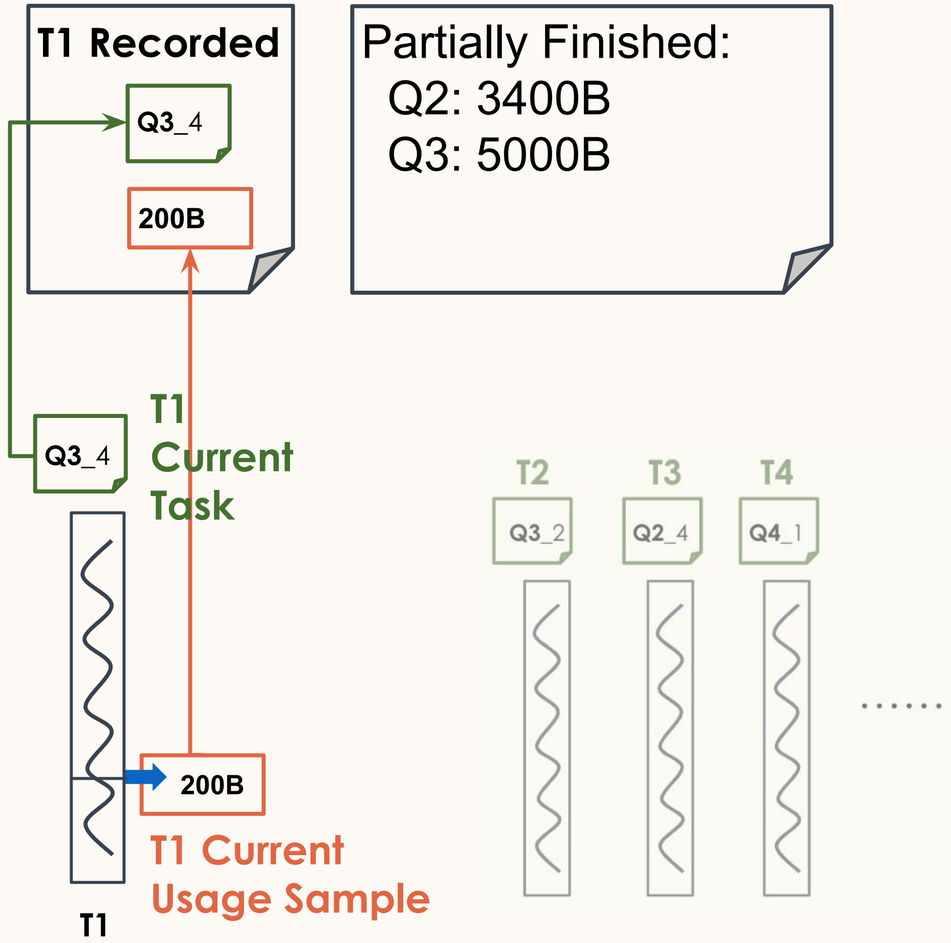




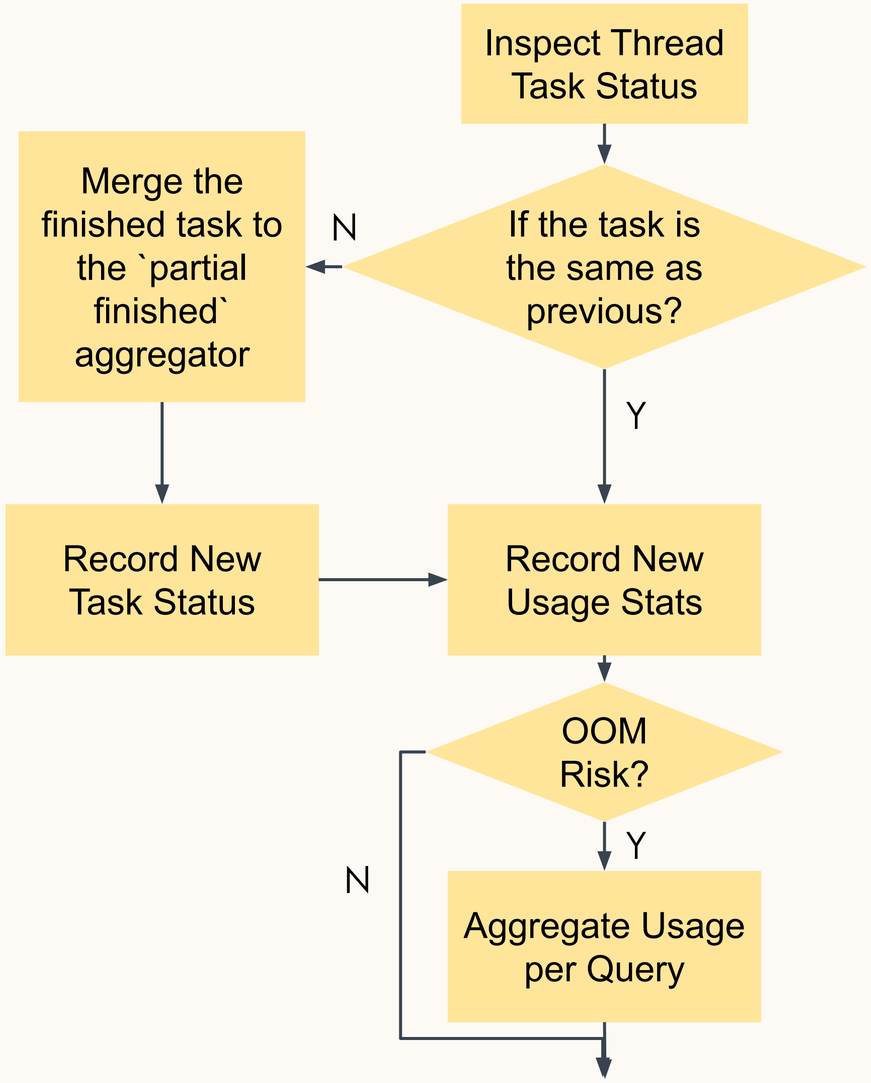
*For simplicity we demonstrate only 1 thread from the threadpool

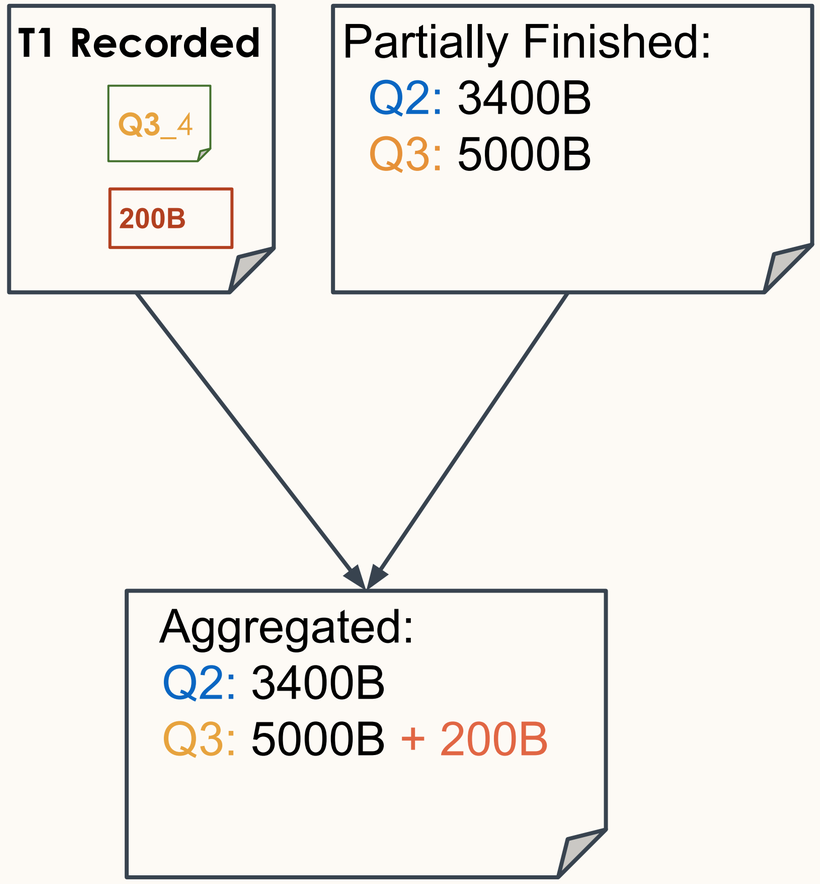
Query Usage Accounting Algorithm





Query Usage Accounting Algorithm

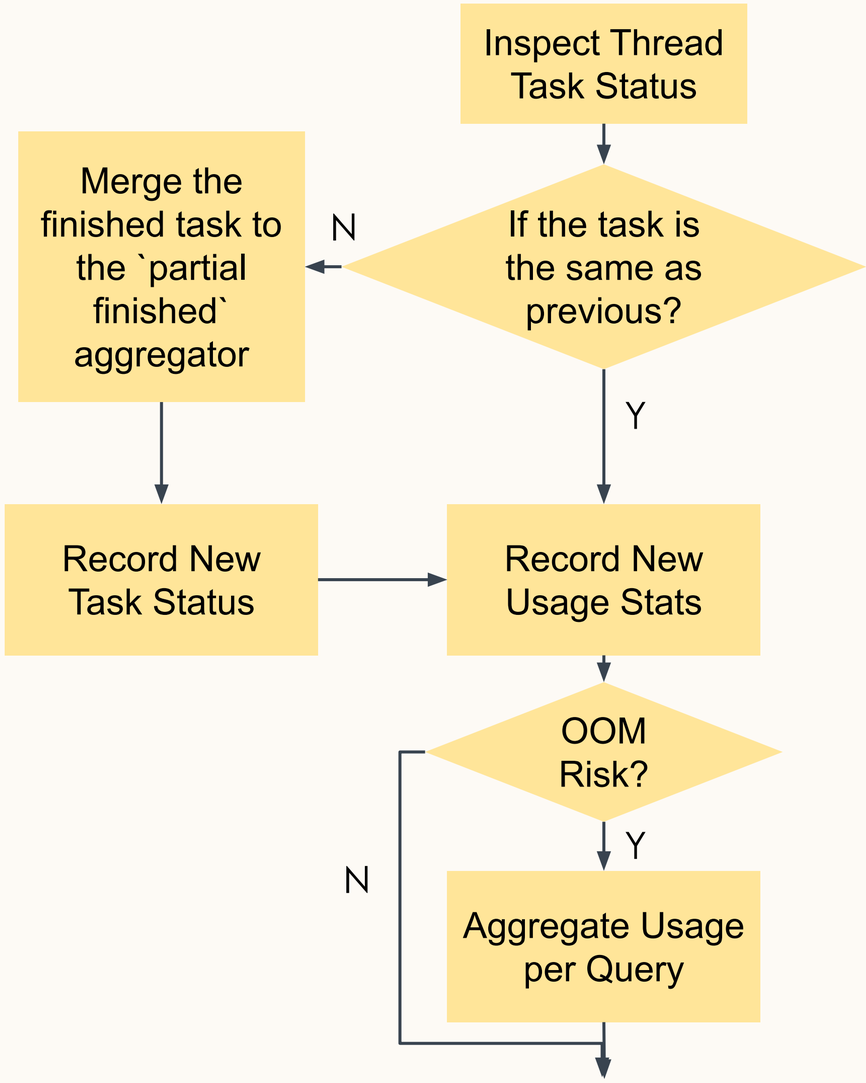




Query Aggregation/Accounting Algorithm

Global Stats Aggregation

- Handles threadpool with fixed/non-fixed threads
- Lives outside of query code path
- Sampling, only tracking big queries Ignoring short lived ones.
- Returns killing code & usage info



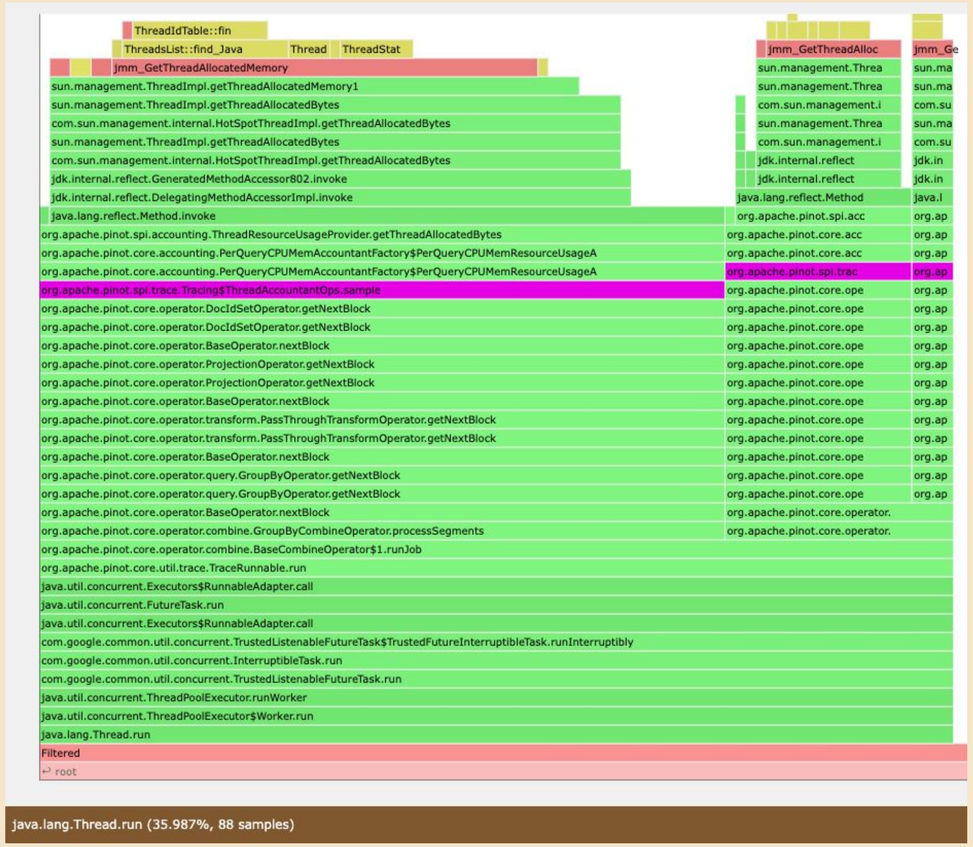
Query Killing

Agenda

1. Motivation and Challenges
2. Design
3. **Results after prod rollout at LinkedIn**

Production Results - Overhead, Observability, Perf Tuning

Negligible Overhead



Overhead = 1% (Filtered) * 35.987% = 0.3%

Observability

- Publishing heap usage. Alert on broker and server `queryKilled` metric
- Internal dashboard filtering the killed and top resource intensive queries from centralized logs and group by unique request ids
- Return the killing messages to customer and give a warning to not retry

Perf Optimization

- G1GC can be quite ‘lazy’ and cause heap usage shootup & long major GC pauses
- Shenandoah GC (SGC) keeps the heap usage lower
- SGC helps with eliminating risk of false positives and potentially improves tail latency

Outcome Highlight

~20

Queries Triggered OOM/quarter

> 90%

Prevented OOM crashes and cascading impact of resource intensive queries by killing more than 85% of such queries

Outcome Highlight

40hrs

Time spent triaging OOMs/quarter

< 4hr

More than 90% toil reduction to
(1) Identify resource intensive queries and
(2) RC OOM crashes, chase culprit queries

Future Work

1. Query Killing: Killing decision propagation
2. Adaptive Server Selection:Enriched stats & enhanced server selection algorithms
3. Fair Scheduling
4. Workload Management

Contributing to Apache Pinot

- We are looking for contributions!
- Apache Pinot 1.0 release is available at https://pinot.apache.org
- Pinot Twitter Account https://twitter.com/ApachePinot
- Pinot Meetup Page https://www.meetup.com/apache-pinot
- Pinot Slack Channel https://tinyurl.com/pinotSlackChannel

Thank you!

Adaptive Server Selection Doc

Query Killing Doc