Query Processing Resiliency

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

Community Over Code ASF Conference 2023
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

REST {QUERY}

REST {ADMIN}

Broker

Realtime Server

Offline Server

Controller

ZK

Helix

Notify

Load Segment

Notify Upload Segment

Segment Store

Ingestion Job

Realtime Ingest

Write Segment

Write Segment

Notify

Data in motion

Apache Kafka,
Azure EventHub
Amazon Kinesis
Google PubSub

NFS,
HDFS,
Azure Storage
Amazon S3
Google Storage

Data at Rest
Remember to add the colored intersection in this core color to your slide, above your image, as Google Slides does NOT support image placeholders.

```java
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, (a1, a2) -> {
            int val = Double.compare(a1.getRight(), a2.getRight());
            return val;
        });

        return pairList;
    }
}
```
Agenda

Adaptive Server Selection

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

- 250K+ Queries Per Second
- 5000+ Pinot Server Hosts
- 1400+ Pinot Broker Hosts
- 4500+ Pinot Tables
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
Building Blocks of the feature

Stats Collector
- Local to each broker
- Stats are stored in-memory
- Per-server stats that are maintained
  - # of in-flight 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-flight 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
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.

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

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

- 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

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

1. Problem Statement
2. Design
3. Regression Benchmarking
4. Results after prod rollout at LinkedIn
Transient Server Slowness
Before
Number of slow server latency alerts per quarter

60

After
Number of slow server alerts per quarter

2
Outcome Highlight

Before
Number of engineering hours spent in debugging transient server slowness issues: 72

After
Number of engineering hours spent in debugging transient server slowness issues: 8
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.
private void killMostExpensiveQuery() {
    if (_aggregatedUsagePerActiveQuery.isEmpty() && _numQueriesKilledConsecutively >= _geTriggerCount) {
        System.gc();
        LOGGER.warn("Triggered gc after killing {} queries", _numQueriesKilledConsecutively);
        _numQueriesKilledConsecutively = 0;
        try {
            Thread.sleep(_normalSleepTime);
        } catch (InterruptedException ignored) {
        }
        _usedBytes = MEMORY_MM_BEAN.getHeapMemoryUsage().getUsed();
        if (_usedBytes < _criticalLevel) {
            return;
        }
    }
    if (_isThreadMemorySamplingEnabled || _isThreadCPSOSamplingEnabled) {
        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;
    }
    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 = _maxHeapQueryEnabled && maxUsageTuple._allocatedBytes > _minMemoryFootprint;
        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
    }
}
Purpose of Layout:

Query Killing

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

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

*Q1_0*
query-task context

Worker Threadpool
Worker Threads

e.g. ThreadMXBean
.getThreadAllocatedBytes()
Generic Status/Usage Reporting for Multi-threaded Query Execution Code

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

Getting Usage Statistics with Instrumentation

*Q1_0 denotes task 0 for query 1

Per Thread Task Status, AtomicReference

Per Thread Stats Sample, Volatile Primitive

e.g. ThreadMXBean .getThreadAllocatedBytes()
Stats Collection - Example

Query Execution Thread (worker thread as an example)
Stats Collection: Example

Setting Up Query-Task Context

```java
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.setWorker(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.addAll(threadResourceUsageProvider.getThreadTimeNs());
            }
        });
    }
}
```
Stats Collection: Example

Instrumenting code

```java
@Override
protected DocIdSetBlock getNextBlock() {
    if (_currentDocId == Constants.EOF) {
        return null;
    }

    // Initialize filter block document Id set
    if (_blockDocIdSet == null) {
        _blockDocIdSet = _filterOperator.nextBlock().getBlockDocIdSet();
        _blockDocIdIterator = _blockDocIdSet.iterator();
    }

    Tracing.ThreadAccountantOps.sample();

    int pos = 0;
    int[] docIds = THREAD_LOCAL_DOC_IDS.get();
    // ...
```

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

Building Accounting/Killing upon Execution Instrumentation

Accounting Thread

Inspect/Record thread context

- OOM Risk?
  - Y
    - Aggregate Usage by Query
    - Terminate the Query With Most Memory
    - Reschedule after X ms
  - N
    - Accountant records thread level usage

Query Execution Thread

Setup Task Status

- Work on a block of data
- Report Usage
- Return Result
- Status?
  - Return With Error
- Kill query threads
- Continue operator execution

Finish
Query Usage Accounting Algorithm

*For simplicity we demonstrate only 1 thread from the threadpool*
Query Usage Accounting Algorithm

- **Partially Finished:**
  - Q2: 3400B
  - Q3: 5000B

- **Current context**
  - **T1**
  - **T1 new context**
  - **T1 Current Usage Sample**

- **Inspect Query-Task context**
- **Context same as previous?**
  - **N**
  - **Y**

- **Record New Task Status**
- **Record New Usage Stats**
- **Merge the finished task to the 'partial finished' aggregator**
Query Usage Accounting Algorithm

- **T1 old context**
  - **Q3**: 5000B + 200B

- **Partially Finished**
  - **Q2**: 3400B
  - **Q3**: 5000B

- **Aggregated**
  - **Q2**: 3400B
  - **Q3**: 5000B + 200B

- **Merge the finished task to the `partial finished` aggregator**
  - **Inspection**
    - **Context same as previous?**
      - **Y** → **Record New Usage Stats**
      - **N** → **Merge the finished task to the `partial finished` aggregator**
  - **Record New Task Status**
  - **OOM Risk?**
    - **Y** → **Aggregate Usage per Query**
    - **N** → **Record New Usage Stats**
Query Usage Accounting Algorithm

*For simplicity we demonstrate only 1 thread from the threadpool*
Query Usage Accounting Algorithm

- **T1 Recorded**
  - Q3.4
  - 200B

- **Partially Finished:**
  - Q2: 3400B
  - Q3: 5000B

- **T1 Current Task**

- **T1 Current Usage Sample**

- **Inspect Thread Task Status**
  - If the task is the same as previous?
    - **Y**
      - Record New Usage Stats
    - **N**
      - Merge the finished task to the 'partial finished' aggregator
      - Record New Task Status
Query Usage Accounting Algorithm

1. Inspect Thread Task Status
2. If the task is the same as previous?
   - Yes: Aggregate Usage per Query
   - No: Record New Task Status
3. If the task is the same as previous?
   - Yes: Record New Usage Stats
   - No: Merge the finished task to the 'partial finished' aggregator
4. OOM Risk?
   - Yes: Partially Finished: Q2: 3400B, Q3: 5000B + 200B
   - No: T1 Recorded: Q3: 4

Aggregated:
- Q2: 3400B
- Q3: 5000B + 200B
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 Aggregation/Accounting Algorithm

- Inspect Thread
- Task Status
- Record New Usage Stats
- If the task is the same as previous?
- Aggregate Usage per Query
- OOM Risk?
- Merge the finished task to the 'partial finished' aggregator
- Record New Task Status
- Y N
Agenda

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

Time spent triaging OOMs/quarter

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

40hrs < 4hr
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