Introduction to the stage of data collection

- The analyzed dataset was taken from the Kaggle site. The content of the dataset is a trace of workloads running on eight Google Borg compute schedulers in May 2019. This trace describes each job submission, scheduling decisions, and resource usage data for the jobs in which the programs were executed.

- The data is based on the evaluation of a program and categorization from May 2011, which has enabled extensive research in the field of up-to-date advancements in job scheduling, tasks, computations, and cloud computing. It has been used to produce hundreds of analyses and studies.

- Since 2011, machines and software have evolved, workloads have changed, and the importance of workload diversity has become more apparent. The new tracking allows researchers to examine these changes. The new dataset includes additional information, such as:

1. **Histograms of CPU usage information for each five-minute interval, not just a single data point.**
2. **Information about allocation sets (shared resource reservations used by jobs).**
3. **Information about job parentage for master/worker relationships, such as MapReduce jobs.**

- Just like the previous tracking, these new tracks also focus on requests and resource usage and do not contain any information about end users, their data, or access patterns to storage systems and other services.

- The tracking data is made available through Google BigQuery to enable advanced analyses without the need for local resources.

- This document describes 1) semantics, 2) data format, and 3) the schema of the tracking of resource usage across multiple Google computing cells.

❖The collected data from Google is a set of machines that are placed in physical racks and connected by a high-bandwidth network. A cell is a set of machines, usually located in a single unit, that shares a common management system responsible for assigning tasks to the machines.

Borg supports two types of resource requests:

1- A job (consisting of one or more tasks) that describes the computations a user wants to run.

2- An allocation set (consisting of one or more allocations, or allocation instances) that describes a resource reservation where jobs can be executed.

A task represents a Linux program, which may include multiple processes and must be executed on a single machine. A job may specify an allocation set in which it should be executed; in that case, each of its tasks runs within an allocation instance from that allocation set (receiving resources). If a job does not specify an allocation set, its tasks will consume resources directly from a machine.

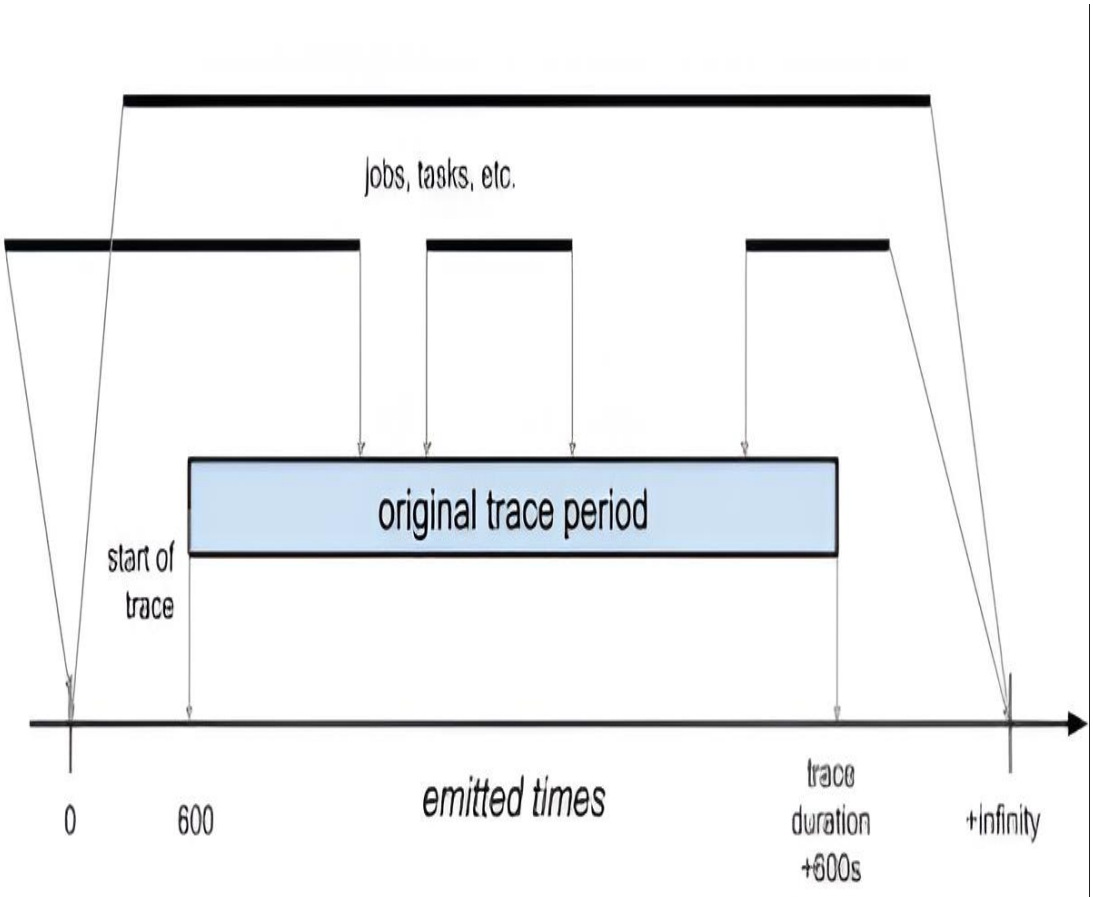
- We refer to jobs and allocation sets as *sets*, and we refer to tasks and allocation instances as *instances*. Throughout the explanations, we will refer to either sets or instances.

- A trace of using a workload on a cell over several days describes it. A trace consists of several tables, each of which is indexed by a primary key, usually containing a **timestamp**. The data in the tables is extracted from the information provided by the cell management system and the individual machines in the cell.

Time and timestamps

- Each record has a timestamp that is calculated in microseconds from 600 seconds before the start of the tracking period and is recorded as a 64-bit integer (for example, an event occurring 20 seconds after the start of the tracking period will have a timestamp of 620 seconds).
- Additionally, there are two special time values that represent events that did not occur within the tracking period.

- **Zero = Time** indicates events that occurred before the start of the tracking period. For example, machines that were present at the beginning of the tracking or something that was sent (or planned) before the start of the tracking.
- **2^{63} - 1 = Time (MAXINT)** indicates events that occur after the end of the tracking period. For example, if a fault occurs in data collection close to the end of the tracking period.



Mapping of original times to times emitted in the trace.

- As an exception, times in usage measurements are handled somewhat differently because the maximum length of measurement is 300 seconds. We apply the same starting time **(time-start)** to these times as we do for events within the tracking period. This is sufficient to provide a clear separation between events before tracking and other events.

- Note that a measurement interval may overlap with the start or end of the tracking period. Additionally, although the actual accuracy for usage measurements is to the nearest second, we still report these times in microseconds for consistency. **Timestamps** are approximate, although reasonably accurate. They are obtained from multiple machines, so small differences in timestamps may indicate a clock change between machines rather than any real scheduling or usage issue.

Unique identifiers

- Each set (task, allocation set) and each machine is assigned a unique 64-bit identifier. These identifiers are never reused. However, machine identifiers may remain the same when a machine is removed from the cell and then returned (accomplish a specific task or procedure). It is very rare for set identifiers to remain the same when the same set is stopped, reconfigured, and restarted. (In our first response, we were aware of only one such example. Usually, restarting a task generates a new job identifier, but the same task name and user name are retained.)

- Tasks are identified by their job ID and a zero-based index within the job. Tasks with the same job ID and task index can (and often do) be stopped and restarted without being assigned a new job ID or index.

- Allocation instances are similarly linked to their associated allocation sets and have a lifecycle similar to that of tasks.

User and collection names

- User and collection names are presented as hashed values and can be tested for equality as base64-encoded strings.

- User and collection names are presented as hashed values and can be tested for equality as opaque base64-encoded strings.

- The logical collection name is a heuristically normalized name for a set of data that combines several internal data field names, hashes the result to create and encode a Base64 string (for example, most numbers in the logical name are replaced with a constant string). Collection names automatically generated by different runs of the same program will typically have the same logical name.

- MapReduce is an example of a Google system that frequently generates unique job names in this manner.
  User names in this tracking represent engineers and services at Google. Production jobs executed by the same user name are likely part of the same external or internal service. Whenever a single-task program runs multiple jobs (for example, a MapReduce program simultaneously creates a main job and a sub-job), these jobs almost always run if they have been submitted by the same user.

Resource units

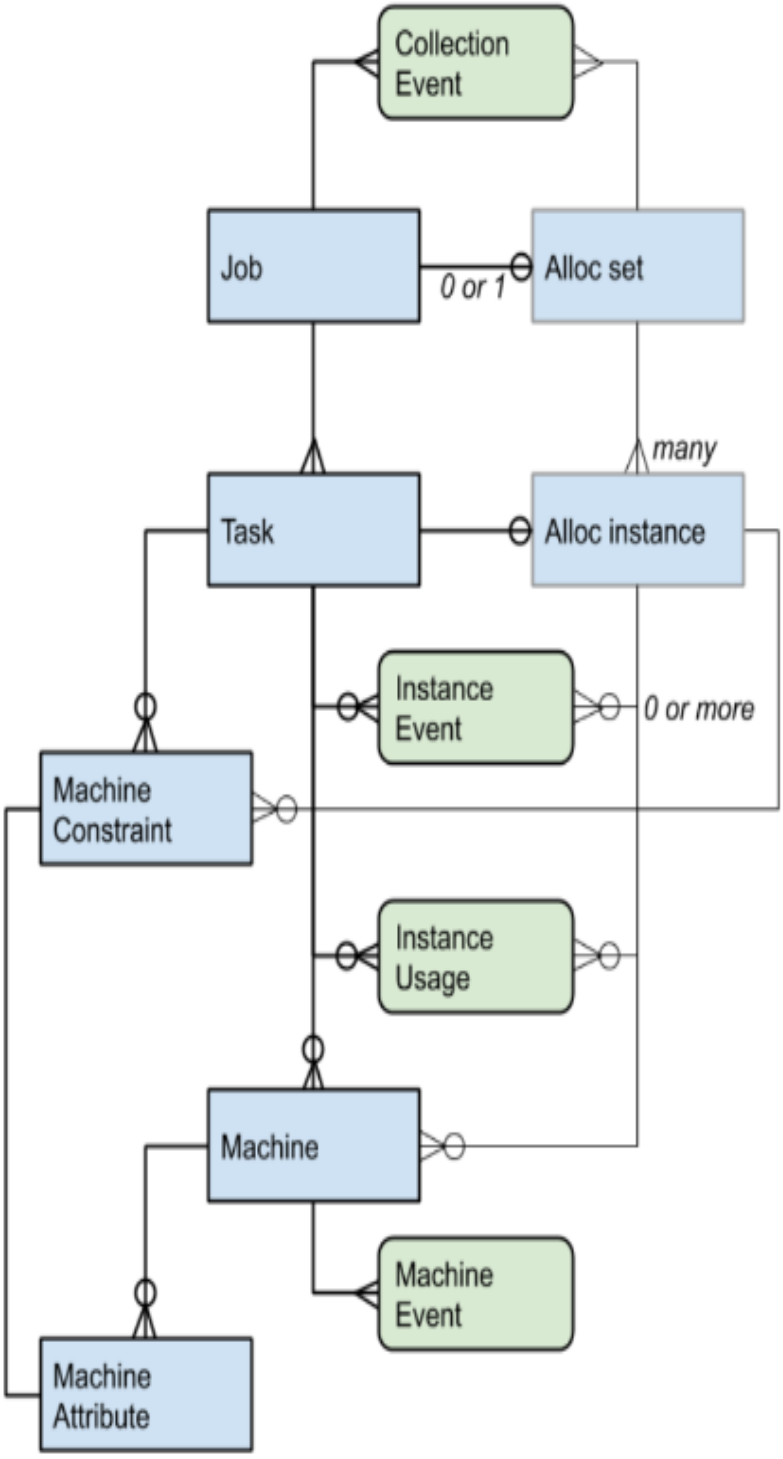
- Resource requests and usage measurements are standardized and scalable as follows:

- Borg measures main memory (RAM) in bytes. We scale memory sizes by dividing them by the maximum memory size observed across all tasks. This means that the capacity of a device with this largest memory size is reported as 1.0. Borg measures CPU in internal units called "Google Compute Units" (GCUs): CPU resource requests are given in GCU units, and CPU consumption is measured in GCU-seconds/second. One GCU represents the computational value of one CPU core on a nominal base machine. The conversion rate of GCU to cores is adjusted for each machine type to provide GCU 1 with approximately the same computational value for our workloads. The number of physical cores allocated to a task to meet its GCU request varies depending on the type of machine the task is mapped to.

- We use a similar scaling for memory: the scaling factor is the largest GCU capacity of the machines in the tracker. In this document, these values are referred to as standard compute units, or NCUs, with values in the range [0,1]. All trackers are standardized using the same scaling factors.

- In practice, most resources are described by a resource structure typically including a value for CPU (in standard compute units, NCUs) and memory (in normalized bytes).

Data tables



An entity-relationship model for the objects (blue) and events (green) found in the traces.

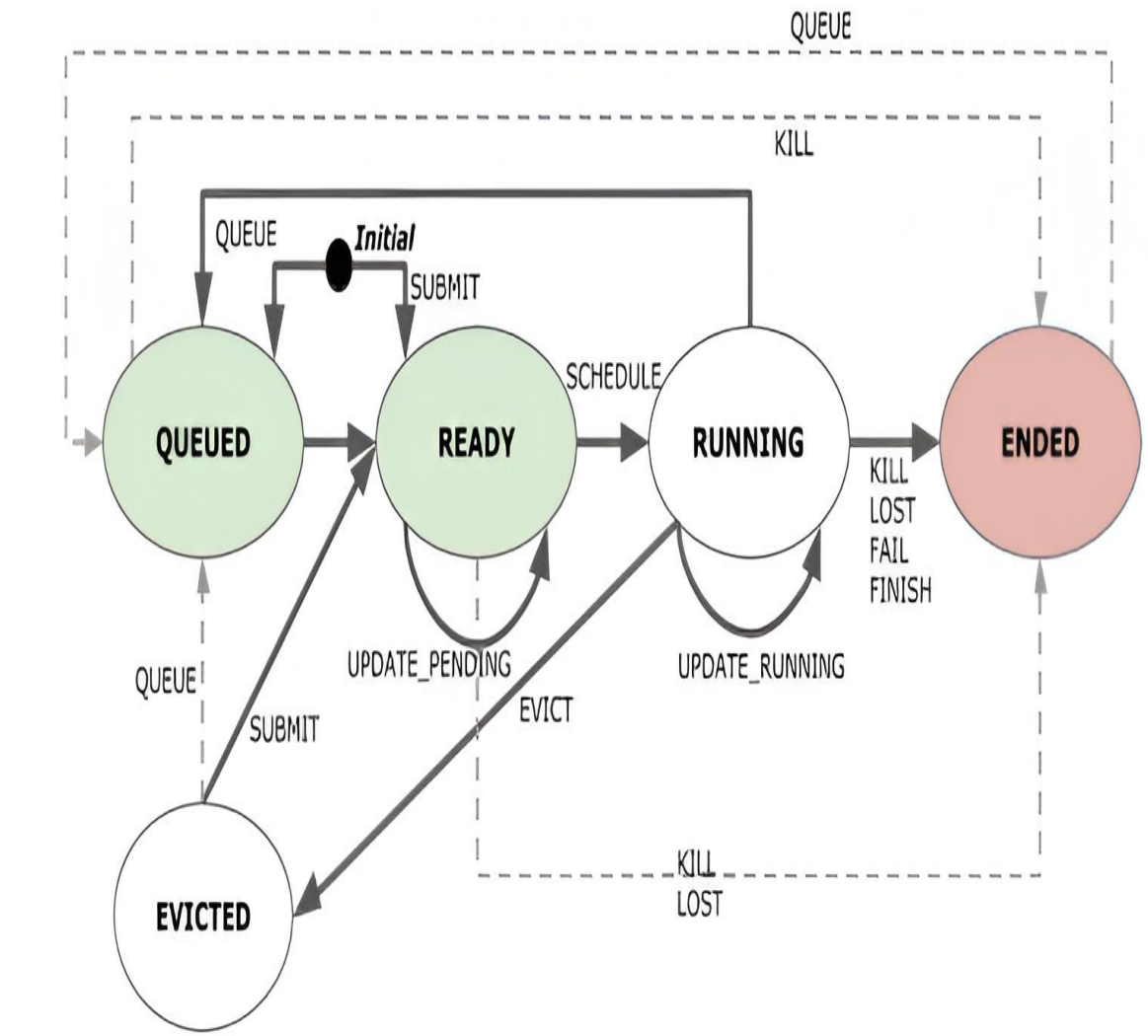
Collections and instances

- The CollectionEvents table and the InstanceEvents table describe the lifecycle of collections and instances, respectively.

- All tasks within a job typically run the same binary with the same options and resource requests. A common pattern for programs requiring different tasks with varying resource needs is to run managers (controllers) and workers in separate jobs (e.g., this method is used in MapReduce and similar systems). A worker job that has a manager job as its parent will automatically stop when the parent terminates, even if the worker is still running. A job can have multiple child jobs but only one parent. Another pattern is to run jobs in a pipeline (this does not apply to allocation sets). If job A is to run after job B, then job A will only be scheduled (set to READY status) once job B has successfully completed. A job

can list multiple jobs that must run before it, and it will only be scheduled after all those jobs have successfully finished. Note that most users who want this capability use external systems outside of Borg to achieve it, so the number of "run after" dependencies is not accurately reflected in the tracking frequency of this behavior in practice.

Collection and instance life cycles and event types



Source state (outgoing transitions >> incoming transitions)

Sink state (outgoing transitions << incoming transitions)

Rare transition (over 100x less frequent than median transition frequency)

Common transition

Expected state transitions for things (collections and instances). The circles represent states; the annotations on the lines are the names of events (transitions). In addition, a small number of additional "unexpected" transitions occur in the traces, typically because of some missing data; to keep the diagram simple, these aren't shown..

Generally, there are two types of events:

1-Events that affect scheduling status (e.g., a job is submitted, or scheduled and becomes runnable, or its resource requests are updated).

2-Events that reflect changes in the status of a task (e.g., a task completes).

Each collection and instance (or object) event has a value that indicates the type of event. The status of the object after the event can always be determined from this type of event. For terminations, the event type also contains information about the cause of termination. Here are the event names (transitions):

- **SUBMIT:** An object was submitted to the section manager.

- **QUEUE:** An object was queued (delayed) until the scheduler is ready to handle it.

- **ENABLE:** An object became eligible for scheduling; the scheduler will try to place the object as soon as possible.

● **SCHEDULE:** An object was scheduled on a machine. (It may not start executing immediately due to code transfer time, etc.) For collections, this occurs when the first instance of them is scheduled on a machine.

● **EVICT:** An object was descheduled due to the higher priority of another object, due to the scheduler's overcommitment and the actual need exceeding the machine's capacity, because the machine on which it was scheduled became unusable (e.g., it went offline for repairs), or due to the loss of the object's data for any reason (this is very rare).

● **FAIL:** An object was descheduled due to some type of user program failure, such as a segmentation fault or a process using more memory than requested from the machine (or in rare cases, it lost its eligibility while it was waiting).

● **FINISH:** An object completed normally.

● **KILL:** An object was canceled by the user or the leading/running program, or its parent object was terminated, or another object that this job depended on finished.

● **LOST:** An object likely finished, but the record indicating its completion was lost in our source data. This event records the moment we realized this issue.

● **UPDATE_PENDING:** The scheduling class, resource requirements, or constraints of an object were updated while it was waiting to be scheduled.

● **UPDATE_RUNNING:** The scheduling class, resource requirements, or constraints of an object were updated while it was running (scheduled).

- The simplest scenario is shown in the upper path of the above diagram: a job is **SUBMITTED** and placed in a queue waiting; immediately afterward, it is **SCHEDULED** on a machine and begins to run; after some time, it **FINISHES** successfully.
- If an object that has been recently **EVICTED**, **KILLED**, or **FAILED** is still executable, a **SUBMIT** event will appear immediately after the unscheduling event—meaning the system tries to restart things that have failed. The actual policy regarding how many times instances can be rescheduled despite abnormal termination varies between collections. For example, though rare, some tasks may be unscheduled indefinitely after an **EVICT** event caused by the scheduler not allocating the requested memory.
- An object may have both **SUBMIT** and **SCHEDULE** events recorded at exactly the same millisecond. This means that an object was submitted and immediately scheduled, or if the timestamp on events is 0, it means that the object was submitted and scheduled before the tracking started.

- Unfortunately, we cannot accurately determine the cause of all instance terminations; in cases where we are unsure, we attribute this to a KILL event, which includes instances where an external program or developer has explicitly taken action to terminate it. In none of these cases is there information about whether the object was running normally or not.

- Note that this is a slightly simplified version of the scheduling system we actually use, and some more complex features (which are hard to explain) have been transferred to this model. We do not believe that any significant accuracy has been lost in this process from the perspective of scheduling research.

Each object has a priority, a small integer that is mapped to an ordered set of values, with 0 being the lowest priority (least important).

Objects with higher priorities generally have preference in accessing resources over those with lower priorities. There are some special priority ranges and specific applications, but generally:

- **Free tier (99 ≥ priorities)**: These are the lowest priorities. The resources requested at these priorities involve minimal internal cost, and there is a weak or no Service Level Objective (SLO) for acquiring and retaining resources on machines.

- **Best-effort category (Best-effort Batch: beb) (priorities 100 - 115)**: Jobs running at these priorities are managed by the batch scheduler and involve minimal internal cost; they do not have any associated Service Level Objective (SLO).

- **Mid-tier (priorities 116 - 119)**: These priorities have Service Level Objectives (SLOs) that fall between the "Free tier" and production priorities.

- **Production tier (Priorities 120 - 359)**: These are the highest priorities in typical usage. In particular, the Borg cluster scheduler tries to avoid evicting latency-sensitive tasks in these priorities due to machine resource overcommitment.

- **Monitoring tier (Priorities ≤ 360):** These priorities are designed for jobs that monitor the health of other lower-priority jobs.

- All jobs and tasks have a scheduling class that roughly indicates how sensitive the task is to latency. The scheduling class is represented by a single number, where a value of 3 indicates a task with higher latency sensitivity (e.g., responding to user requests that generate revenue), and a value of 0 indicates a non-production task (e.g., development, non-critical business analytics, etc.).

- Note that scheduling class is not the same as priority, although latency-sensitive tasks tend to have higher priorities. Priority determines whether an object is scheduled on a machine, while the scheduling class affects the machine's local policy for resource access (its local "quality of service") after scheduling.

- The collection name is hashed and presented as an opaque base64-encoded string, which can be tested for equality with other names. Unique job names are sometimes generated by automated systems to avoid conflicts (MapReduce is an example of a Google system that does this). The collection logical name is an abstract name that combines data from multiple internal name fields (for example, most numbers in the logical name are replaced with a fixed string). Logical names partially compensate for the unique names generated by automated tools; in this case, different runs of a program will usually have the same logical name.

- Resource requests indicate the maximum amount of CPU or memory that an instance is allowed to use (we call this its limit). Tasks that exceed their limit may be throttled (for resources like CPU) or killed (for resources like memory). Since the scheduler may decide to overcommit resources on a machine, there may not be enough resources to meet all runtime requests from tasks, even if each of them is using less than their limit. If this happens, one or more lower-priority tasks may be killed.

- In addition to the explanations provided, the machine's runtime environment sometimes allows tasks to use more than what is specified in the resource request. For example, tasks are permitted to use the machine's free CPU capacity, so tasks with brief CPU bursts and low latency sensitivity (insensitive tasks with brief latency) can be effectively executed with a requested CPU allocation of zero.

Resource usage

- Our machines use Linux containers for resource isolation and usage accounting. Each task runs in a separate container and may spawn several processes within that container. Dedicated instances are also associated with a container in which task containers are placed.
- We report usage values from a series of non-overlapping measurement windows for each instance. These windows are typically 5 minutes (300 seconds) long, although they may be shorter if the instance starts, stops, or is updated during that time period. Measurements may continue for several seconds after an instance ends, or, rarely, for a few minutes.