Google cluster-usage traces v3

John Wilkes, with much help from Charles Reiss, Nan Deng, Md Ehtesam Haque, and Muhammad Tirmazi.
Original version 2020-04-01, updated 2020-05-01, 2020-07-28, 2020-08-10, 2020-08-18.

Table of contents

Introduction

This document describes the semantics, data format, and schema of usage traces of a few Google compute cells. This document describes version 3 of the trace format. We will assume the reader is familiar with the following two papers:

- Large-scale cluster management at Google with Borg. Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, John Wilkes. *Proc. European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.
- Borg: the Next Generation. Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, John Wilkes. *Proc. European*

Conference on Computer Systems (EuroSys), Heraklion, Crete, Greece, 2020.

A Google *cluster* is a set of machines, packed into physical enclosures (racks), and connected by a high-bandwidth cluster network. A *cell* is a set of machines, typically all in a single cluster, that share a common cluster-management system that allocates work to machines.

Borg supports two kinds of resource request: a *job* (made up of one or more *tasks*), which describes computations that a user wants to run, and an *alloc set* (made up of one or more *allocs*, or *alloc instances*), which describes a resource reservation that jobs can be run in. A task represents a Linux program, possibly consisting of multiple processes, to be run on a single machine. A job may specify an alloc set in which it must run, in which case each of its tasks will run in (receive resources from) an alloc instance of that alloc set. If a job doesn't specify an alloc set, its tasks will consume resources directly from a machine. We call jobs and alloc sets *collections*, we refer to tasks and alloc instances as *instances*, and use *thing* to refer to collections or instances.

A single usage trace describes several days of the workload on a single Borg cell. A trace is made up of several *tables*, each indexed by a primary key that typically includes a timestamp. (See below for the details of how the tables are packaged for access.) The data in the tables is derived from information provided by the cell's management system and the individual machines in the cell.

Common techniques and fields

This section describes the representation and semantics of fields that occur in several tables. Note that a trace may not include all the types of data described here.

Obfuscation techniques

For confidentiality reasons, we have obfuscated certain information in the trace. In particular, most free-text fields have been randomly hashed, resource sizes have been linearly transformed (scaled), and certain values have been mapped onto a sorted series. We took care to do this in a consistent way so that the data can still be used in research studies.

Here are the obfuscation transformations that are used for most data types:

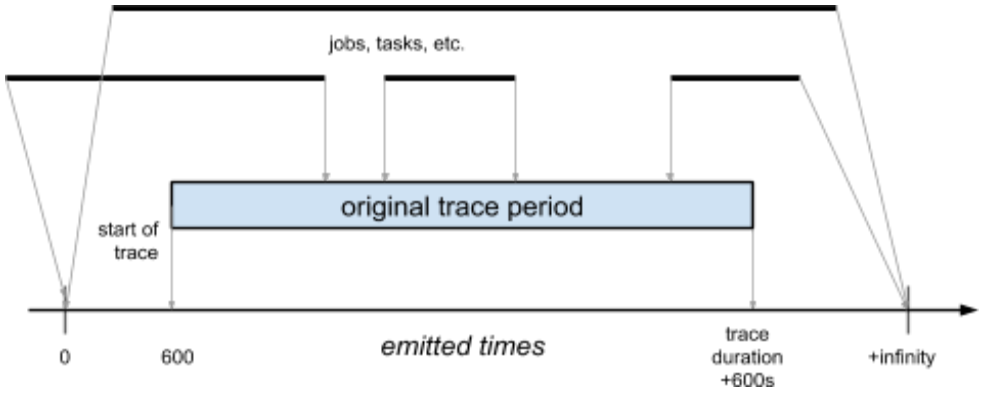
- *unchanged*: the values are not obfuscated.
- *hashed*: each value is transformed to an opaque value by means of a keyed cryptographic hash.
- *ordered*: the list of observed (or possible) values is generated and sorted. Then, the items in this list are assigned sequential numbers starting with 0 (i.e., gaps are elided), and the observed values are mapped onto these numbers before adding them to the trace.
- *rescaled*: each value is transformed by dividing by a type-specific constant so that values are in the range [0,1]. The resulting normalized value is then rounded to a granularity of $max(1.0/2^{10}, max\_value/2^{20})$ - i.e., no more than ten bits of binary precision or ~1 millionth of the maximum value, whichever is larger. Almost all floating point values are transformed in this way, using the same rescaling factors across all of the v3 traces.
- *special*: a few values are treated in a special way, described below.

Time and timestamps

Each record has a timestamp, which is in microseconds since 600 seconds before the beginning of the trace period, and recorded as a 64 bit integer (i.e., an event 20 seconds after the start of the trace would have a timestamp=620s).

Additionally, there are two special time values representing events that did not occur within the trace window:

- A time of 0 represents events that occurred before the beginning of the trace window. Examples: machines that existed at the start of the trace or a *thing* that was submitted (or has already been scheduled) before the start of the trace.
- A time of 2^{63}-1 (MAXINT) represents events that occur after the end of the trace window (e.g., if there is a data collection failure near the end of the trace window).



Mapping of original times to times emitted in the trace.

As an exception, times in usage measurements are treated slightly differently, because the maximum measurement length is 300 seconds. We apply the same start-time offset to these times as we do for events inside the trace window; this is enough to ensure a clear separation between events before the trace and other events. Note that a measurement interval may overlap the beginning or end of the trace. Additionally, although the actual precision 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 sourced from multiple machines, and so small disagreements in timestamps may reflect clock drift between machines in the cluster instead of any real scheduling or utilization issue.

Unique identifiers

Every collection (job, alloc set) and every machine is assigned a unique 64-bit identifier. These IDs are never reused; however, machine IDs may stay the same when a machine is removed from the cluster and returned. Very rarely, collection IDs may stay the same when the same collection is stopped, reconfigured and restarted. (We were aware of only one such example in our first trace. Usually, restarting a job will generate a new job ID but keep the same job name and user name.)

Tasks are identified by the job ID of their job and a 0-based index within the job. Tasks with the same job ID and task index can be (and frequently are) stopped and be restarted without assigning a new job ID or index. Alloc instances are similarly tied to the alloc set of which they are a part, and have a similar lifecycle to tasks.

User and collection names

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

The *logical collection name* is a heuristically normalized name of the collection that combines data

from several internal name fields, and hashes the result to construct an opaque base64-encoded string (e.g., most numbers in the logical name are replaced with a fixed string). Auto-generated collection names generated by different executions of the same program will usually have the same logical name. MapReduce is an example of a Google system that frequently generates unique job names in this manner.

Usernames in this trace represent Google engineers and services. Production jobs run by the same username are likely to be part of the same external or internal service. Whenever a single program runs multiple jobs (for example, a MapReduce program spawns both a master job and worker job), those jobs will almost always run as if they were submitted by the same user.

Resource units

Resource requests and utilization measurements have been normalized and scaled, as follows.

Borg measures main memory (RAM) in bytes. We rescale memory sizes by dividing them by the maximum machine memory size observed across all of the traces. This means that the capacity of a machine with this largest memory size will be reported as 1.0.

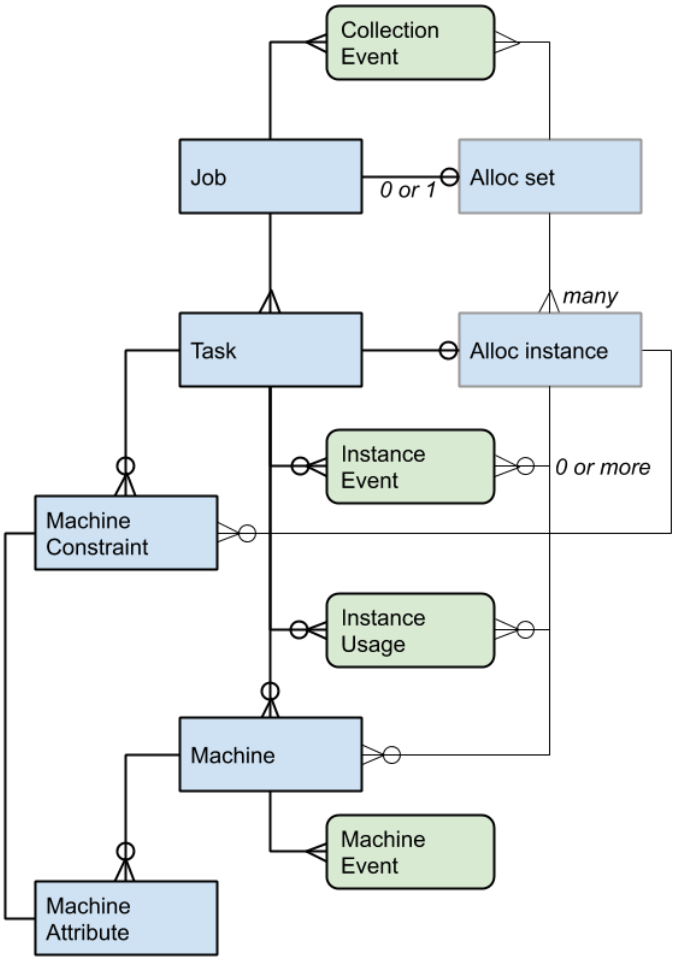
Borg measures CPU in internal units called "Google compute units" (GCUs): CPU resource requests are given in GCUs, and CPU consumption is measured in GCU-seconds/second. One GCU represents one CPU-core's worth of compute on a nominal base machine. The GCU-to-core conversion rate is set for each machine type so that 1 GCU delivers about the same amount of useful computation for our workloads. The number of physical cores allocated to a task to meet its GCU request varies depending on the kind of machine the task is mapped to. We apply a similar scaling as for memory: the rescaling constant is the largest GCU capacity of the machines in the traces. In this document, we refer to these values as *Normalized Compute Units*, or *NCUs*; their values will be in the range [0,1]. All of the traces are normalized using the same scaling factors.

In the trace, most resources are described by a Resources structure, which typically contains a value for both CPU (in NCUs) and memory (normalized bytes).

Data tables

This section describes the tables in the traces. Table keys are *italicized*; note that some keys are constructed from multiple fields. The master definition of the fields in the tables lives in the `clusterdata_trace_format_v3.proto` file on GitHub; you can think of these descriptions as an annotated commentary on that file.

The following entity-relationship model may be helpful.



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

Machines

Machines are described by two tables: the *MachineEvents table* and the *MachineAttributes table*.

Machine events

Each machine is described by one or more records in the machine event table. The majority of records describe machines that existed at the start of the trace.

MachineEvents table:

1. *time*
2. *machine_id*
3. type – the type of event (see below)
4. switch_id – (see below)
5. capacity – the resources that the machine supplies to programs that run on it; typically smaller than the physical machine's raw capacity (a Resources structure)
6. platform_id – (see below)
7. missing_data_reason – one of:
    a. NONE: no data was missing
    b. SNAPSHOT_BUT_NO_TRANSITION: we know that a change to the state of the

machine must have happened, but we did not see the corresponding machine event

There are three types of machine events:

- ADD: a machine became available to the cluster – all machines in the trace will have an ADD event; many will occur before the start of the trace proper, at time=0.
- REMOVE: a machine was removed from the cluster. Removals can occur due to failures or maintenance; a subsequent ADD event for the same machine generally signals the latter.
- UPDATE: a machine available to the cluster had its available resources changed.

Machine capacities reflect the normalized physical capacity of each machine along each dimension. Each dimension (CPU cores, RAM size) is normalized independently. Note that not all of this capacity is available to tasks; for example, the local Borglet agent needs to reserve some resources for itself and the operating system.

The platform ID is an opaque string representing the microarchitecture and chipset version of the machine. Different combinations of micro-architectures (e.g., vendor chip code-names) and memory technologies (e.g. DDR2 versus DDR3) will result in different platform IDs. Two machines with the same platform ID may have substantially different clock rates, memory speeds, core counts, etc.

The switch ID is a unique identifier for the network switch that the machine is connected to (commonly called a "top of rack switch"); machines connected to the same switch are usually in the same rack (physical enclosure). There may be more than one switch per rack.

Machine attributes

Machine attributes are key-value pairs representing machine properties, such as the kernel version, clock speed, presence of an external IP address, etc. Instances can specify constraints on machine attributes (see machine constraints).

MachineAttributes table:

1. *time*
2. *machine_id*
3. *name* – the obfuscated attribute name
4. value – either an obfuscated (hashed) string or an integer
5. deleted – a boolean indicating whether the attribute is being deleted

Machine attribute values are provided as integers if there are a small number of values for that attribute name or as obfuscated (hashed) strings otherwise. In the first case, the values of the attributes observed across all machines are recorded and sorted (in numerical order, if all are numbers). The first value is mapped to 1, the second to 2, and so on.[1] Some attributes have no associated values because their presence or absence is the signal (e.g., whether a machine is running a particular program); such attributes are given the value 1 if present.

Collections and instances

The *CollectionEvents table* and *InstanceEvents table* describe the life cycle of collections and instances respectively.

All the tasks within a job usually execute the same binary with the same options and resource request. Programs that need to run different types of tasks with different resource requirements

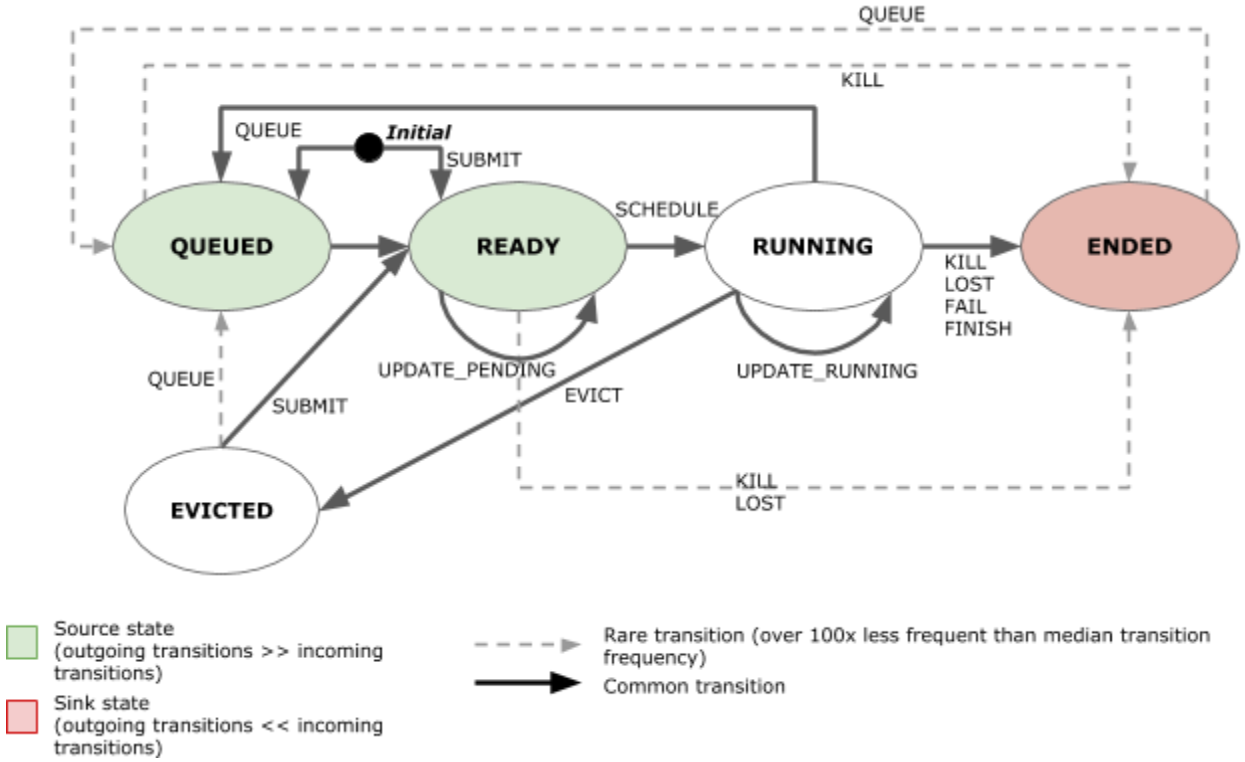
[1] This approach was also used in "Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters", Victor Chudnovsky, Rasekh Rifaat, Joseph Hellerstein, Bikash Sharma, Chita Das, *Symposium on Cloud Computing (SoCC)*, Oct. 2011.

usually execute as separate jobs.

A common pattern is to run masters (controllers) and workers in separate jobs (e.g., this is used in MapReduce and similar systems). A worker job that has a master job as a *parent* will automatically be terminated when the *parent* exits, even if its workers are still running. A job can have multiple child jobs but only one *parent*.

Another pattern is to run jobs in a pipeline (this doesn't apply to alloc sets). If job A says that it should *run after* job B, then job A will only be scheduled (made READY) after job B successfully finishes. A job can list multiple jobs that it should *run after*, and will only be scheduled after all those jobs finish successfully. Note that most users who desire this functionality use systems outside Borg to achieve it, so the number of *run after* dependencies in the trace doesn't accurately reflect the frequency of this behavior in practice.

Collection and instance life cycles and event types



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..*

There are basically two types of events: ones that affect the scheduling state (e.g., a job is submitted, or it gets scheduled and becomes runnable, or its resource requests are updated), and ones that reflect state changes of a task (e.g., the task exits).

Each collection and instance (or *thing*) event has a value representing the type of event. The state of the thing after the event can always be determined from this event type. For deaths, the event type also contains information about the cause of the death. Here are the event (transition) names:

- SUBMIT: a thing was submitted to the cluster manager.
- QUEUE: a thing is queued (deferred) until the scheduler is ready to act on it**.**
- ENABLE: a thing became eligible for scheduling; the scheduler will try to place the thing as soon as it can.
- SCHEDULE: a thing was scheduled on a machine. (It may not start running immediately due to code-shipping time, etc.) For collections, this occurs the first time *any* of its instances is scheduled on a machine.
- EVICT: a thing was descheduled because of a higher priority thing, because the scheduler overcommitted and the actual demand exceeded the machine capacity, because the machine on which it was scheduled became unusable (e.g., taken offline for repairs), or because the thing's data was lost for some reason (this is very rare).
- FAIL: a thing was descheduled (or, in rare cases, ceased to be eligible for scheduling while it was pending) due to a user program failure of some kind such as a segfault, or a process using more memory than it requested.
- FINISH: a thing completed normally.
- KILL: a thing was cancelled by the user or a driver program, or the thing's parent exited, or another thing on which this job was dependent ended.
- LOST: a thing was presumably ended, but a record indicating its termination was missing from our source data. The event captures the moment when this was realized.
- UPDATE_PENDING: a thing's scheduling class, resource requirements, or constraints were updated while it was waiting to be scheduled.
- UPDATE_RUNNING: a thing's scheduling class, resource requirements, or constraints were updated while it was running (scheduled)**.**

The simplest case is shown by the top path in the diagram above: a job is SUBMITted and gets put into a pending queue; soon afterwards, it is SCHEDULEd onto a machine and starts running; some time later it FINISHes successfully.

If a thing that has just been EVICTed or KILLed, or has FAILed, remains runnable, a SUBMIT event will appear immediately after the descheduling event – i.e., the system tries to restart things that have failed. The actual policy about how many times instances can be rescheduled in spite of non-normal termination varies between collections. For example, though rare, it is possible for some tasks to be indefinitely descheduled after an EVICT event that was caused by the scheduler not giving the task its requested memory.

A thing may have both SUBMIT and SCHEDULE events for the exact same microsecond in the trace. This means that a thing was submitted and immediately scheduled, or, if the timestamp on the events is 0, that the thing was submitted and scheduled before the trace began.

Unfortunately, we are not able to precisely determine the cause of all instance terminations; where we are uncertain, we map this to a KILL event, which also includes cases where an external program or a developer took explicit action to terminate it. There is no information about whether a thing was executing normally in either case.

Note that this is a slightly simplified version of the scheduling system we actually employ, and a few of the more sophisticated (*aka* hard to explain) features have been mapped down into this model. We do not believe that any significant fidelity has been lost in this process from the point of view of resource-scheduling research.

Missing event records

Our data sources contain both time-series of thing changes and periodic snapshots of thing state, so we are able to check for consistency between them. When it seems we are missing an event record, we synthesize a replacement. Similarly, we look for a record of every thing that is active at the end

of the trace time window, and synthesize a missing record if we don't find one.

Synthesized records have a missing_type field with a non-zero value to represent why they were added to the trace:

- SNAPSHOT_BUT_NO_TRANSITION: we did not find a record representing the given event, but a later snapshot of the thing state indicated that the transition must have occurred. The timestamp of the synthesized event is the timestamp of the snapshot.
- NO_SNAPSHOT_OR_TRANSITION: we did not find a record representing the given termination event, but the thing disappeared from later snapshots of cluster states, so it must have been terminated. The timestamp of the synthesized event is a pessimistic upper bound on its actual termination time assuming it could have legitimately been missing from one snapshot.
- EXISTS_BUT_NO_CREATION: we did not find a record representing the creation of the given thing. In this case, we may be missing metadata (job name, resource requests, etc.) about the thing and we may have placed SCHEDULE or SUBMIT events later than they actually are.
- TRANSITION_MISSING_STEP: some event must have been missed for this state transition to have occurred; most often associated with an instance being rescheduled onto a different machine.

Records without missing data have no value (i.e., an empty string) in the "missing_info" field.

CollectionEvents table

Every event in this table describes an event associated with a collection (job or alloc set). The fields are as follows. The first set is common to collections and instances, although the interpretation is slightly context-sensitive.

1. *time*
2. type – see the list above
3. *collection_id* – a unique identifier for this collection across all jobs and alloc sets within the same trace (they share the same ID space)
4. scheduling_class – see below
5. missing_type – see Missing event records
6. collection_type – 0=job, 1=alloc set
7. priority – a small integer; larger value means higher priority (see below)
8. alloc_collection_id - (jobs only) the unique identifier of the alloc set that hosts the job, or 0 if this is a top-level collection.

Fields specific to collections:

9. user – the obfuscated name of the "user" (person or system) that submitted the collection
10. collection_name – a hash of the original complete collection name
11. collection_logical_name – a hash of the parts of the collection name that reflect its purpose, and excluding things like sequence numbers or UIDs
12. parent_collection_id - the unique identifier of the parent collection, or 0 if it has none. The collection will be killed if/when its parent terminates.
13. start_after_collection_ids - zero or more unique identifiers of any collections that must finish before this one can start.
14. max_per_machine – the preferred maximum number of instances from the collection that may be placed on the same machine, or 0 if there is no limit
15. max_per_switch – the preferred maximum number of instances from the collection that may be placed on machines that share a common top-of-rack network switch, or 0 if there is no limit
16. vertical_scaling – if enabled, the system determines how much CPU and RAM to request for the collection rather than requiring the user to specify them manually. Values are:
    - VERTICAL_SCALING_SETTING_UNKNOWN: we were unable to determine what policy

was used (rare)
    - VERTICAL_SCALING_OFF: autoscaling was not enabled
    - VERTICAL_SCALING_CONSTRAINED: autoscaling was enabled with user-supplied conditions/parameters
    - VERTICAL_SCALING_FULLY_AUTOMATED: autoscaling was enabled with no user-provided bounds
17. scheduler – the scheduler that was tasked with placing the thing. Values are:
    - SCHEDULER_DEFAULT: the default job scheduler
    - SCHEDULER_BATCH: a secondary (batch) scheduler

The thing event tables include any things that are active (RUNNING) or eligible to run but waiting to be scheduled (PENDING) at any point in the trace. For every collection in the trace, we will include at least one record for all its instances, which will include its scheduling constraints.

Each thing has a *priority,* a small integer that is mapped here into a sorted set of values, with 0 as the lowest priority (least important). Things with larger priorities (higher "tiers") generally get preference for resources over things with smaller priorities. There are some special priority ranges, and uses, but in general:[2]

- *Free tier* (priorities ≤ 99): these are the lowest priorities. Resources requested at these priorities incur little internal charging, and have weak SLOs or no SLOs for obtaining and keeping resources on machines.
- *Best-effort Batch (beb)* (priorities 100–115): jobs running at these priorities are managed by the batch scheduler and incur low internal charges; they have no associated SLOs.
- *Mid-tier* (priorities 116–119): these priorities have SLOs that are between the "free" ones and the production priorities.[3]
- *Production tier* (priorities 120–359): these are the highest priorities in normal use. In particular, the Borg cluster scheduler attempts to prevent latency-sensitive tasks at these priorities from being evicted due to over-allocation of machine resources.
- *Monitoring tier* (priorities ≥ 360): these priorities are intended for jobs that monitor the health of other, lower-priority jobs.

All jobs and tasks have a *scheduling_class* that roughly represents how latency-sensitive the task's execution is. The scheduling class is represented by a single number, with 3 representing a more latency-sensitive task (e.g., serving revenue-generating user requests) and 0 representing a non-production task (e.g., development, non-business-critical analyses, etc.). Note that scheduling class is *not* a priority, although more latency-sensitive tasks tend to have higher priorities: the priority determines whether a thing is scheduled on a machine, while the scheduling class affects the machine-local policy for resource access (its local "quality of service") once it has been scheduled.

The *collection_name* is hashed and provided as an opaque base64-encoded string that 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 opaque name that combines data from several internal name fields (e.g., most numbers in the logical name are replaced with a fixed string). Logical names partially compensate for the unique names generated by automatic tools; in this case, different executions of the same program will usually have the same logical name.

---

[2] These numbers are the same as in the Borg: the Next Generation paper, except for the range for best-effort batch, which was mistakenly reported there as 110-115. That paper also contains the mapping of the priority bands used in the 2011 trace to these tiers.

[3] For an example, see Marcus Carvalho, Walfredo Cirne, Franciso Brasileiro, John Wilkes Long-term SLOs for reclaimed cloud computing resources, *ACM Symposium on Cloud Computing (SoCC)*, 2014, 20:1-20:13.

InstanceEvents table

This table provides information about events for instances (tasks and alloc instances). The first set of fields are the same as for the CollectionEvents table (see above):

1. *time*
2. type – the type of the collection that this instance is a part of
3. *collection_id* – the collection that this instance is a part of
4. scheduling_class
5. missing_type
6. collection_type
7. priority
8. alloc_collection_id (tasks only), or 0 if not running in an alloc set or if this is an alloc instance

Fields specific to instances:

9. *instance_index* – position within the collection
10. machine_id – the machine on which the instance was scheduled or 0 if it is not; the special value –1 means a dedicated machine (we lump them all together)
11. alloc_instance_index – the index of the alloc instance that the task is running in, or –1 if not part of an alloc
12. resource_request – the CPU and memory resources requested for the instance (a Resources structure)
13. constraint – zero or more placement constraints (see Machine constraints below)

The resource requests represent the maximum amount of CPU or memory an instance is permitted to use (we call this its *limit*). Tasks using more than their limit may be throttled (for resources like CPU) or killed (for resources like memory). Because the scheduler may choose to over-commit resources on a machine, it can happen that there are not enough resources to meet all the runtime requests from the tasks, even though each of them is using less than its limit. If this happens, one or more low priority task(s) may be killed.

Additionally, the machine's runtime environment sometimes permits tasks to use more than what is specified in the resource_request. For example, tasks are permitted to use free CPU capacity on the machine, so tasks with brief latency-insensitive CPU bursts might usefully run with a requested CPU allocation of 0.

An instance may have *machine constraints*, which are hard placement constraints against machine attributes that restrict the machines on which the instance can run. There may be multiple constraints on a single attribute. (For example an instance *could* have both a LESS_THAN_EQUAL and a GREATER_THAN_EQUAL constraint for the same attribute.) This information is provided in the *constraint* column in the *Instance Events* table, in the form of a *Machine Constraints* structure that contains the following fields:

1. name – an obfuscated (hashed) name of a constraint
2. value – either an obfuscated (hashed) string, or an integer, or the empty string
3. relation – a comparison operator. The constraint matches the machine (i.e., permits the instance to be scheduled there) if:
    - EQUAL: the value of the machine *name* attribute treated as a string (or the empty string if the attribute is not present) matches the constraint's *value*
    - NOT_EQUAL: as for EQUAL, but the strings must not match
    - LESS_THAN: the value of the machine *name* attribute treated as an integer (or 0 if the attribute is not present) is *strictly less than* the constraint's *value*
    - LESS_THAN_EQUAL, GREATER_THAN, GREATER_THAN_EQUAL: as for LESS_THAN, but with a different comparison operation
    - PRESENT: the machine has a *name* attribute, regardless of its value

- NOT_PRESENT: the machine does not have a *name* attribute

Resource usage

Our clusters use Linux containers for resource isolation and usage accounting. Each task runs within its own container and may create multiple processes in that container. Alloc instances are also associated with a container, inside which task containers nest.

We report usage values from a series of non-overlapping measurement windows for each instance. The windows are typically 5 minutes (300s) long, although may be shorter if the instance starts, stops, or is updated within that time period. The measurements may extend for up to tens of seconds after an instance is terminated, or (rarely) for a few minutes.

During each measurement window (of duration T_{window}), we sample usage data roughly once per second. We aggregate the values in these samples as follows:

- CPU usage: on each sample, we query the OS kernel for the CPU-seconds consumed (U_{cpu}) by an instance during the length of the sample (T_{sample}). And then … (all units are CPU usage second per second).
  - The average CPU usage rate for a time window is $\Sigma(U_{cpu}) \ / \ T_{window}$.
  - The maximum CPU usage rate over a time window is $\max(U_{cpu}/T_{sample})$.
  - We also compute percentiles for the window using the U_{cpu} values.
  - All raw data is normalized to NCUs.
- Memory usage (U_{mem} in bytes) is collected every sampling period.
  - The average memory consumption for a window is $\Sigma(U_{mem} \times T_{sample}) \ / \ T_{window}$ – i.e., the area under a graph of memory-usage vs time divided by the window duration.
  - The maximum memory usage over a time window is $\max(U_{mem})$.
  - All raw data is normalized.

A few measurement records may be missing because of limitations in our reporting systems. Missing records do not necessarily indicate that a task was not running.

There are some task usage measurements for periods when no process belonging to the task was running in the task's container. For example, these measurements might occur while binaries are being copied to the machine. In this case, memory and CPU usage for a task may legitimately be 0. In some cases, a task may be in state RUNNING with no process for an extended period of time.

The resource usage reported for an alloc instance is the usage for all the tasks running within it during each sampling period.

The ***InstanceUsage table*** contains these fields:

1. *start_time* of the measurement period
2. *end_time* of the measurement period
3. *collection_ID*
4. *instance_index* – within the collection
5. machine_id
6. alloc_collection_id – or 0 if this instance is not running inside an alloc
7. alloc_instance_index – the value is undefined if the instance is not running inside an alloc
8. collection_type – job or alloc set
9. average_usage – the average usage (see above) across the portion of the window in which the instance is scheduled (a Resources struct)
10. maximum_usage – largest observed usage during the window; it may be omitted or 0 in some cases (a Resources struct)
11. random_sampled_usage – the observed usage during a randomly-selected 1s sample in the

window; CPU data only (a Resources struct). For long-running tasks, this data can be used to build a reasonably accurate stochastic model of CPU usage.
12. assigned_memory – the average memory limit (upper bound) for this instance given to the OS kernel by the Borglet
13. page_cache_memory – the average memory used for the instance's file page cache by the OS kernel
14. cycles_per_instruction – the mean CPI during the window (obtained by counting the CPU cycles used and dividing by the number of instructions executed)
15. memory_accesses_per_instruction – the mean MAI during the window (obtained by counting the memory accesses used and dividing by the number of instructions executed)
16. sample_rate – the number of samples taken per second during the window. The nominal target is 1 Hz, but system load may cause this to be lower. (For example, if samples are only taken every 2s, the sample_rate would be 0.5.)
17. cpu_usage_distribution – 11 coarsely-spaced percentiles of the observed CPU usage during different samples: 0%ile (minimum), 10%ile, 20%ile, 30%ile, 40%ile, 50%ile, 60%ile, 70%ile, 80%ile, 90%ile, 100%ile (maximum) Note that
18. tail_cpu_usage_distribution – 9 finely-spaced percentiles of the observed CPU usage during different samples: 91%ile, 92%ile, 93%ile, 94%ile, 95%ile, 96%ile, 97%ile, 98%ile, 99%ile

CPU usage (also known as CPU rate) is measured in units of NCU seconds per second: if a task is using two GCUs all the time, it will be reflected as a usage of 2.0 GCU-s/s before normalization.

The *cpu_usage_distribution* and *tail_cpu_usage_distribution* vectors provide detailed information about the distribution of CPU consumption during the 5 minute measurement window, in NCUs. For example, the NCU usage for the 90%ile represents the CPU consumption that 90% of the CPU usage samples during the window would be equal to or smaller than (i.e., we are providing values from the cumulative distribution function, or CDF). Because these percentiles are interpolated values, there may be small discrepancies with other data.

Since memory isolation is achieved through Linux memcg, some kernel memory usage on behalf of the task is accounted to the task. Tasks must request sufficient memory to include such allocations. The following memory usage data are included:

- memory: memory usage based on the memory actually assigned to the container (but not necessarily used)
- maximum memory usage: the maximum memory usage measurement observed over the measurement interval. This value is not available for some tasks.

Cycles Per Instruction (CPI) and Memory Accesses Per Instruction (MAI) statistics are collected from processor performance counters; not all machines collect this data. Memory accesses are based on measurements of last-level cache misses.

Accessing the trace data

We provide the traces as BigQuery tables and JSON files.

BigQuery access

Each trace is a separate dataset, with a name of the form `2019-05-nn`, in the project `Google Cluster Data` (id=`google.com:google-cluster-data`). For example, the *machine events table* for the 2019-05-a trace will be found in `google.com:google-cluster-data:clusterdata_2019_a.machine_events`.

Important: the trace data is freely available, but using BigQuery will consume project resources that may have to be paid for. We strongly encourage researchers to keep an eye on their resource consumption, and to sample the data when developing queries.

Here are some simple examples of using the BigQuery command line tool to query the data set:

- Checking the data set in cluster *a*:
  `bq show google.com:google-cluster-data:clusterdata_2019_a.machine_events`

- Counting the number of unique machines in cluster a in machine_events table:
```
bq query --use_legacy_sql=false \
'SELECT COUNT(DISTINCT machine_id)
FROM `google.com:google-cluster-data`.clusterdata_2019_a.machine_events'
```

JSON access

In addition to BigQuery access, the trace is also available for download in JSON format from Google cloud storage (GCS).

Each cell's data is stored in its own bucket whose name follows the pattern `clusterdata_2019_${CELL}`. (E.g., the data of cell `a` is stored in the bucket `clusterdata_2019_a`.)

Inside each bucket, there are 5 tables: `collection_events`, `instance_events`, `machine_events`, `machine_attributes` and `instance_usage`. Each table is sharded into one or more files, whose names follow the pattern `${TABLE_NAME}-[0-9]+.json.gz`, where the number following the table name represents the shard's id. For example, the instance usage data for cell `a` is stored at `gs://clusterdata_2019_a/instance_usage-*.json.gz`, where `*` is a wildcard.

To download the data, one can use gsutil:

```
gsutil cp gs://clusterdata_2019_a/instance_usage-*.json.gz <destination dir>
```

To inspect the content of a file, one needs to first decompress the data using GZIP. E.g.:

```
gunzip instance_usage-000000000000.json.gz
```

Each line in the file contains a JSON object representing a row in the table. Note that because JSON does not support int64 type, any int64 field (e.g. timestamps, *collection_id, machine_id*) are represented as strings in JSON format. Here is an example of one row of data from machine_events table:

```
{"time":"89703182129","machine_id":"375997113395","type":"1","switch_id":"0kdfKLeqkk1sN
8xXnJ8f63bMq+ciUu2ztSu53+pf1HM=","platform_id":"JQ1tVQBMHBAIISU1gUNXk2powhYumYA+4cB3KzU
2918="}
```

The schema of each table is stored in another GCS bucket named `clusterdata_2019_schema`. Inside the bucket, there are 5 files each representing the schema of their corresponding table using JSON (e.g., `collection_events.schema.json`). The schema files provide effectively the same information as `clusterdata_trace_format_v3.proto`. For example, the first few lines of `collection_events.schema.json` are as follows:

```
[
  {
    "type": "INTEGER",
    "name": "time",
```

```
    "mode": "NULLABLE"
  },
  {
    "type": "INTEGER",
    "name": "type",
    "mode": "NULLABLE"
  },
  {
    "type": "INTEGER",
    "name": "collection_id",
    "mode": "NULLABLE"
  },
...
```

Missing information

This data is primarily derived from monitoring data, collected by periodic remote procedure calls (RPCs) inside the Borg ecosystem. When the monitoring system or cluster gets overloaded, data may not be collected. For scheduler events, we have synthesized records to make the data consistent as described above, but some data is still likely to be missing from the trace.

Much of the underlying raw data is collected from the Linux OS kernel; some data may be missing or even incorrect, especially under overload conditions.

Our cluster scheduler supports several features that are not represented in the traces. These include:

- Some additional resource types and their usage are not recorded.
- "Preferred" or “soft” constraints, which express a preference, but not a hard constraint, are not included here. Such preferences may be violated by the scheduler when it places instances.
- Workloads not managed by the scheduler: some systems run outside the control of the cluster scheduler, and the task usage and machine capacity measurements do not account for these workloads.

Dedicated machines

Some of our workloads run on “dedicated” machines in the clusters, which are machines that are allocated to specific users. Since our focus was on the scheduling behavior for the shared workload, we made the following adjustments to the raw Borg trace data:

- dedicated machines are omitted from the *MachineEvents table*
- collections that run entirely on dedicated machines are omitted from the *CollectionEvents table*, and their instances are omitted from the *InstanceEvents table*
- collections that run partly on dedicated machines are included in the *CollectionEvents table* and *InstanceEvents table*, but when one of their instances is placed on a dedicated machine, the machine_id used is a special one meaning “some dedicated machine”
- there are no entries in the *InstanceUsage table* for instances running on dedicated machines

Traces

This section provides detailed information about the currently-available traces. The version 3 format was developed to support the publication of several traces for the month of May 2019. This

is the set of traces that are available in that format.

Trace	Trace start (t=600s)	Timezone
2019-05-a	1 May 2019 00:00 PDT	America/New York
2019-05-b	1 May 2019 00:00 PDT	America/Chicago
2019-05-c	1 May 2019 00:00 PDT	America/New York
2019-05-d	1 May 2019 00:00 PDT	America/New York
2019-05-e	1 May 2019 00:00 PDT	Europe/Helsinki
2019-05-f	1 May 2019 00:00 PDT	America/Chicago
2019-05-g	1 May 2019 00:00 PDT	Asia/Singapore
2019-05-h	1 May 2019 00:00 PDT	Europe/Brussels

Document history

Date	*Notes*
2020-04-01	First published version
2020-04-22	Corrected names of BigQuery tables, and added example queries.
2020-05-01	Clarified the description of the machine constraint structure.
2020-08-10 (2020-08-18)	Updated the priority tier descriptions and numbers. Added JSON format. (Fixed a few typos.)