Incident Review: What Comes Up Must First Go Down

Incident review facilitated by Nathan Lincoln,
write-up by Fred Hebert

honeycomb.io

Table of contents

On July 25th, 2023, we experienced a total Honeycomb outage spanning ingest, querying, triggers, SLOs, and our API. The condensed timeline is that following 10 minutes of partially degraded ingestion, we saw a rapid failure cascade that took down most of our services. The outage impacted all user-facing components from 1:40 p.m. UTC to 2:48 p.m. UTC, during which no data could be processed or accessed.

The user interface recovered around 2:48 p.m. UTC, and querying recovered unevenly across all partitions and teams. During this time, requests may have contained outdated information, or simply failed.

Ingestion came back up around 3:15 p.m. UTC, at which point we could accept incoming traffic and API requests. Query capacity, along with triggers and SLO alerting, kept regaining accuracy and succeeded for more and more users, until service was fully restored at 3:35 p.m. UTC.

Our total outage time was roughly one hour and 10 minutes, with an extra 28 minutes for ingest (10 minutes partial, 18 minutes total), and 47 minutes of degraded querying and alerting (triggers and SLOs) for roughly two hours of incident time.

This outage is our biggest (or most total) since we've had paying customers, but the events behind it are unremarkable. In this review, we will cover the incident itself, and then we'll zoom back out for an analysis of multiple contributing elements. Finally, we'll go over our response and the aftermath.

The incident

The setup

The events started on Monday July 24th, late in the day for our west coast engineers, when we switched between two clusters of Retriever (our storage and query engine). We found a potential bug on the cluster that represents the next software version to run there, so we decided to go back to the old version and fix the problem on the new, unused cluster in the morning. This is a routine change we've done multiple times in the past, and it felt like the safest option.

Shortly after, our internal SLO for Shepherd (our ingest service) started slowly burning. This happened in short spikes at the end of each hour; however, since the service doesn't normally experience heavy load at night, it took multiple hours to make a significant enough dent for our engineers to be notified.

A different group of engineers looked into the issue, which seemed localized: the calls to refresh their local in-memory cache appeared slow, and a live profiling check pointed at contention around some mutexes. Eventually, they came to the conclusion that the database itself was slow, which bubbled out to the ingest service. There had been a need for ingest scale-up around that time, but the write volume didn't match either. They looked into new code that shipped earlier in the day and found nothing that could explain a database or ingest issue.

At about 8:00 p.m. local time, the engineers decided that since the problem was just a marginally slow, inconsistent performance burn with a lack of good explanation, this was a minor issue that could be investigated in the morning.

This is a judgment call we often recommend to engineers to ensure they are well rested when dealing with incidents.

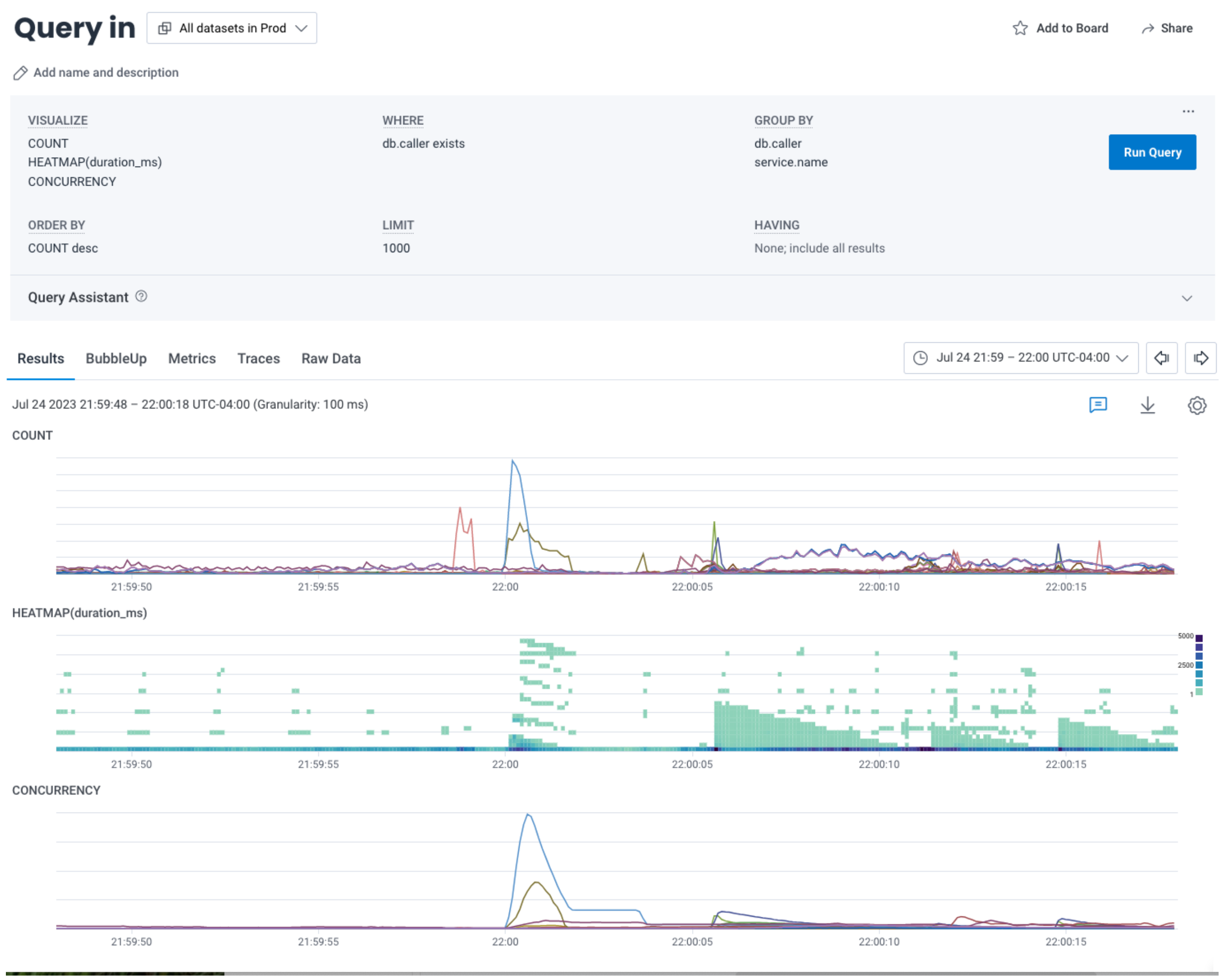
Early morning pages

Around 4:00 a.m. PDT, our on-call engineers—different from those who switched flags or who investigated the night before—received pager alerts for ingest issues and performance. This time around, the alerts paged because of the morning east coast scale-up in ingest volume, which amplified the performance issues.

As the engineers looked into the alerts and the Slack logs from the night before, they noticed it was a continuation of the same problem. They, too, concluded that this was an intermittent issue without significant customer impact that could wait for investigation during business hours.

Business hours investigation

In the morning, during east coast working hours, one of our engineers saw the issue logs from the night before. They went over the same threads others had looked into, saw a query that looked at all the database calls all of our services did, and re-ran it on a wider timeline (24 hours instead of one minute) to see if any pattern held up over time. This pointed out a big change:



Query in All datasets in Prod

Add to Board Share

Add name and description

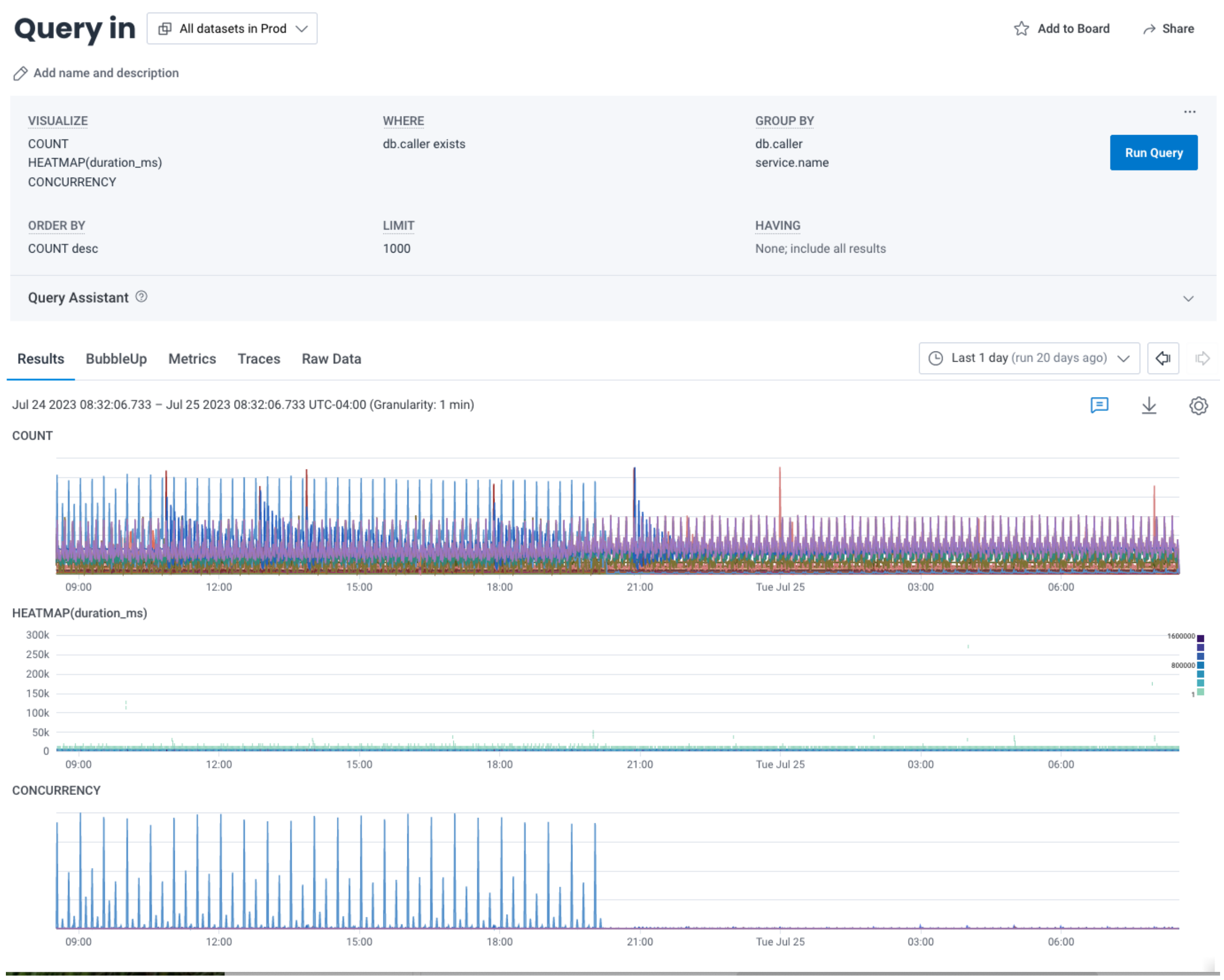
VISUALIZE	WHERE	GROUP BY
COUNT HEATMAP(duration_ms) CONCURRENCY	db.caller exists	db.caller service.name

ORDER BY	LIMIT	HAVING
COUNT desc	1000	None; include all results

Query Assistant

Jul 24 2023 21:59:48 – 22:00:18 UTC-04:00 (Granularity: 100 ms)

Query showing the database calls over a one minute window, hoping to highlight specific costly operations that stack up at the time symptoms are observed.



Query in All datasets in Prod

Add name and description

VISUALIZE
COUNT
HEATMAP(duration_ms)
CONCURRENCY

WHERE
db.caller exists

GROUP BY
db.caller
service.name

ORDER BY
COUNT desc

LIMIT
1000

HAVING
None; include all results

Jul 24 2023 08:32:06.733 – Jul 25 2023 08:32:06.733 UTC-04:00 (Granularity: 1 min)

The same query over a 24 hour period, hoping to show wide trends in events over time to highlight changes.

As a side note, this is a practice our engineers recommend: going from a narrow to a wide view often rapidly invalidates or confirms potential investigation paths by showing whether patterns are specific to the current investigation, or normal and misleading.

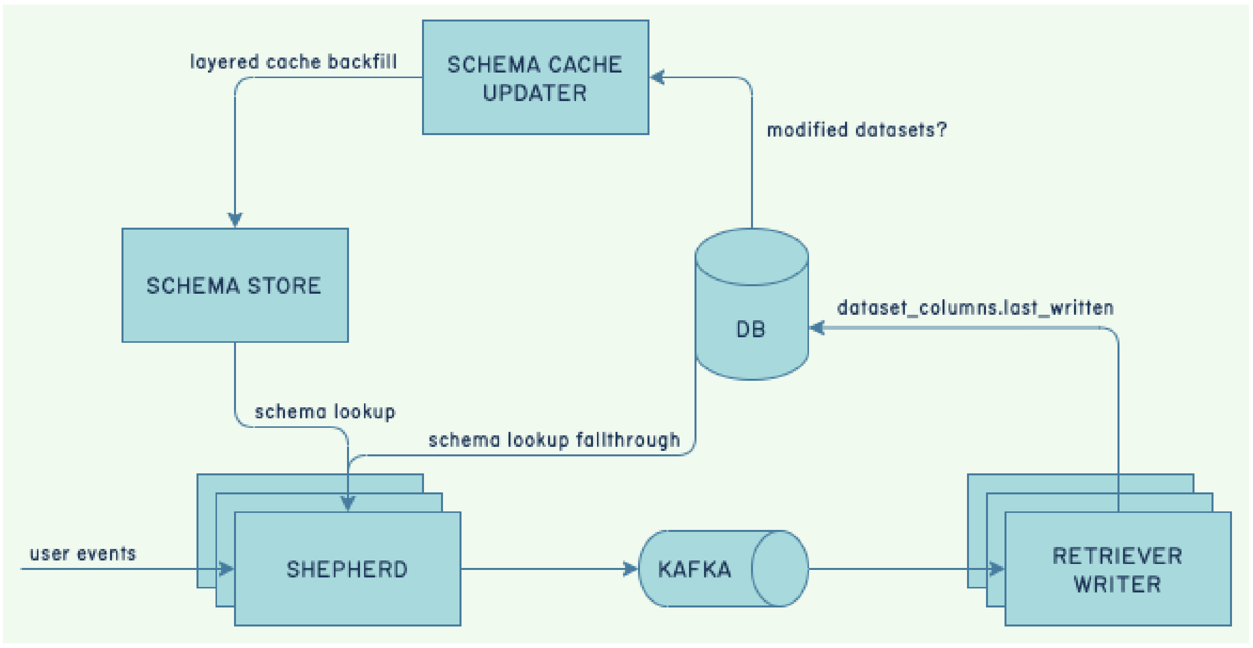
Going from a narrow view to a wide one immediately revealed a drastic drop in some database calls (the blue ones). These calls are named SetDatasetColumnsLastWritten and they come from our Retriever service, and updates the last time we received data for any field, for any dataset, for any customer.

Two things stood out right away:

1. This change was related to the shift in infrastructure from the night before. The engineer who noticed the pattern was aware of the change, and knew that shifting between clusters also shifted which set of servers would update these fields.
2. This field was used by our caching mechanism to know which dataset schemas we needed to actively refresh, and was used to pre-warm caches for the whole ingest service.

Put together, these things provided a convenient explanation: something about these updates on the old query engine cluster failed, which undermined the cache. This explained the ingest performance issues.

The dependency cycle looks a bit like this:



High-level architecture diagram showing how user events are used by the writer to update schemas, and that field is required for cache backfilling, which if not done, has read fall through to the database.

The issue still wasn't considered critical at this point, since there was only a slight degradation in performance. Shepherds maintain their own in-memory cache, and we assumed that the latency spikes happened when they bypassed the schema store to run expensive queries, but otherwise, things were stable. This was still weird, though. We've made the switch between storage clusters multiple times before without a problem. So as our engineers came online according to their respective time zones, they looked into what could be behind this drop of writes to bring them back.

Since there's a slight variation in infrastructure between the clusters, one of the theories we entertained was that maybe, through recent changes, some permissions or write mechanisms had gone wrong on the old infrastructure. We changed which cluster handled the writes back to the new one by flipping a feature flag. This would let us get the fields updated without running the risk of having queries encounter the bug we still had to address.

We've used that flag many times before, so we knew that flipping it temporarily stops all the writes: the timestamp for any individual entry is only updated every 10 minutes or so, and switching the writes creates a synchronization point by resetting timers uniformly. It usually takes roughly 10 minutes before seeing them come back at full volume. After these 10 minutes, however, they still didn't come back. Things started to feel even weirder. We restarted a couple hosts, but didn't notice an improvement.

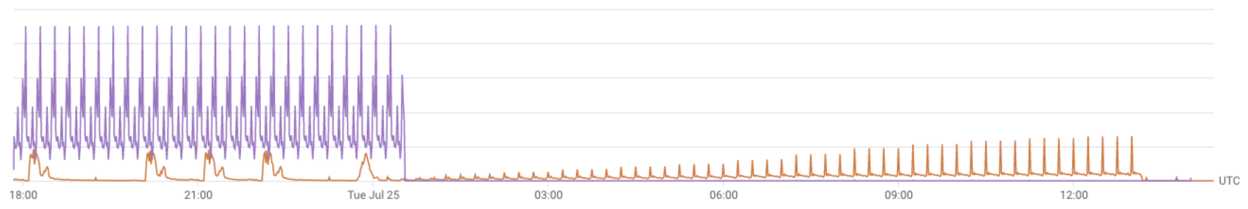
An engineer started to dig into the feature flag's implementation, and noticed a subtle bug: whenever we switch the flag, the goroutine that writes the updates returns instead of continuing over the current iteration. Therefore, whichever hosts did the writing in a timed loop stopped doing so, but more importantly, never tried again even if the flag was switched back.

The key distinction was the time scales and when we used the flag. A full reboot was required for writes to migrate over to a cluster whose writes had been turned off before. The flaw was present in the flag all along, but hidden through our deployment mechanism often running in parallel.

Since deploys caused all hosts on all services to restart, and deploys often happened right around the time we switched flags, it papered over the behavior and made the flag look like it worked as expected. Everyone knew it worked fine.

This surprised people in the incident; some were familiar with this code, had modified or reviewed it in the past, and that "exit and never come back" behavior wasn't something that they ever noticed. As we learned through the incident review process, literally no one in the organization, including the people who wrote the code and used it the most, had any idea that this is how it actually worked.

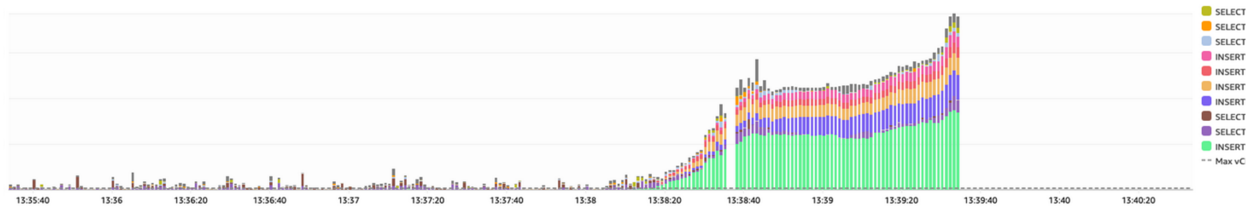
This insight let us find more data. The following query shows the history of the attempted writes per cluster (purple being new, orange being old):



You can see every deployment prior to July 25th (in UTC) causing bumps of attempted writes on the orange line, and then stopping as the code loops exits. The purple cluster is fully used. However, when we switched clusters on July 24th, we did so at the tail end of a deployment window and most but not all the Retriever hosts stopped writing. For the entire night, the few trailing hosts on the old cluster wrote their data, so the customers who send us the most traffic and span many partitions were able to keep their own schemas' cache warm.

When we switched writes across clusters to bring data back in the morning, we stopped the writes and kept the cache from refreshing. The moment we understood this, we wrote a command to roll our query engine's cluster as quickly as possible without interfering with other critical features (querying and alerting).

But mere minutes before we got to it, things went bad:



Query volume before the incident, and after, when all the work started piling up.

The big green line is new schema changes lagging and piling up, which is a very bad thing. They should be going much faster, and in lower volume.

Within two minutes, our end-to-end alerts (which monitor our ability for writes to go through the system and get queried) fired and paged us, and a massive onslaught of database connection errors (for having too many connections) happened. We noticed all critical services were 100% down.

Being in a bad place

In these two minutes, we went from a leisurely morning investigation into a full-blown incident response.

We knew from past experience that ingest outages likely require weird manual circuit-breaking of all traffic just to come back; the hosts can't come up on their own. In this case, we also proactively set up these circuit breakers and denied all traffic with a 5xx error, because we assumed the database issue came from too many schema updates while overloaded.

The plan, in short, was to bring ingest back. But, bringing ingest back without a cache would make it go down again, either through overload or database connection saturation. We first had to make sure the database was ready to take connections, then restarted all of the Retriever hosts on the query engine to provide the data for the cache.

As we cut traffic off, a group of engineers went to see if they could save the database by removing whatever transactions were stuck. If we could take a few extra minutes to salvage the current host, the overall recovery would be faster.

This, however, was unproductive. We couldn't kill connections as everything would hang—even commands like `SHOW ENGINE INNODB STATUS`, which call to none of the usual tables, would never return. It appeared that a confluence of heavy read load, increasing writes, and overall stress in the database locked it up.

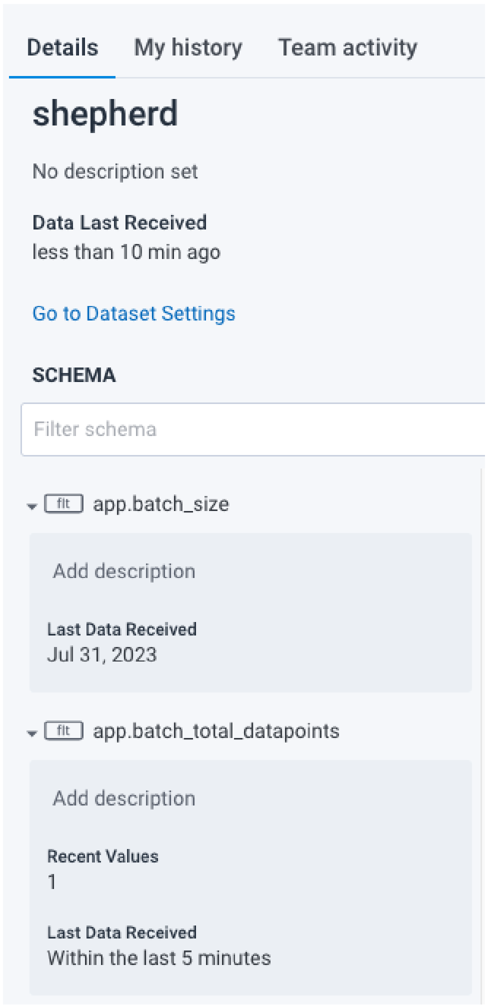
At this point, we decided to call it quits on killing connections to save the database; we weren't making progress in reasonable time. Instead, we decided to fail over to a replica.

As it turns out though, the database came right back up. Most of our services came back nearly instantly. Front-end queries worked, Shepherds (the ingest service) were booted up, and our query engine appeared mostly functional, it just had no fresh data.

But we couldn't bring traffic back just yet by removing our circuit-breaking. The flags were flipped the previous night, which meant freshness data behind the cache was sparse, and since no updates at all had happened since the start of the actual outage, the cache was empty. Retriever was up and could mark recently written data, but wasn't getting traffic: allowing traffic to Shepherd without a cache is risky, and until it flows through, we can't update the cache.

We split into two investigative paths: one of making the schema cache service reload a longer history of data, and one to manually do SQL surgery in our database to force a data reload. Unfortunately, the last experiment we had run with the schema reloading service to make it force reload more data was unsuccessful, and the engineers who owned the service were not up yet.

Since we made progress on the SQL surgery front, we decided to go with that approach. We manually updated the 'last written' timestamp of all schemas that had seen traffic in the last day and marked it as `now`. This introduced incorrect data that is used in two places: the parts of our UI that tell you how fresh data is, and the query assistant, which looks into time windows greater than a day and would have been unaffected during the time of the incident.



shepherd

No description set

Data Last Received
less than 10 min ago

Go to Dataset Settings

SCHEMA

Filter schema

flt app.batch_size

Add description

Last Data Received
Jul 31, 2023

flt app.batch_total_datapoints

Add description

Recent Values
1

Last Data Received
Within the last 5 minutes

Example data for which we introduced inaccuracies in order to bring service back.

This worked, and on the next scan, the schema cache updater reloaded all the data we needed. At that point, we still didn't know how well the database might take the traffic or if it needed to warm up, and a prior outage revealed a tendency for our users to have lots of data buffered that they'd send all at once. We used short time periods with our circuit breaker to let in traffic, then choked it back, and analyzed.

We let traffic through for a few seconds, then turned it off. CPU and memory usage were high, which we believed to be normal when first warming up. After a pause to let it all settle, we let more traffic go through for half a minute or so, and this time it looked stable. We managed to confirm that Retrievers were updating the last updated schema values, and so all the ingredients were in place for a full success. We let traffic go through in a third circuit-breaking reset, which repaired ingest.

We then worked to restore querying capacity. It seemed partially functional, and one of our engineers restarted hosts on a rotation while these efforts were ongoing to make sure the flag was active and that everything was up. Some hosts were lagging behind and required extra restarts since the database faults made them fail some internal checks.

At this point, the whole fleet was healthy, and all we had left to do was to keep restarting the inactive Retriever hosts and investigate the more specific failure modes we've seen in our database.

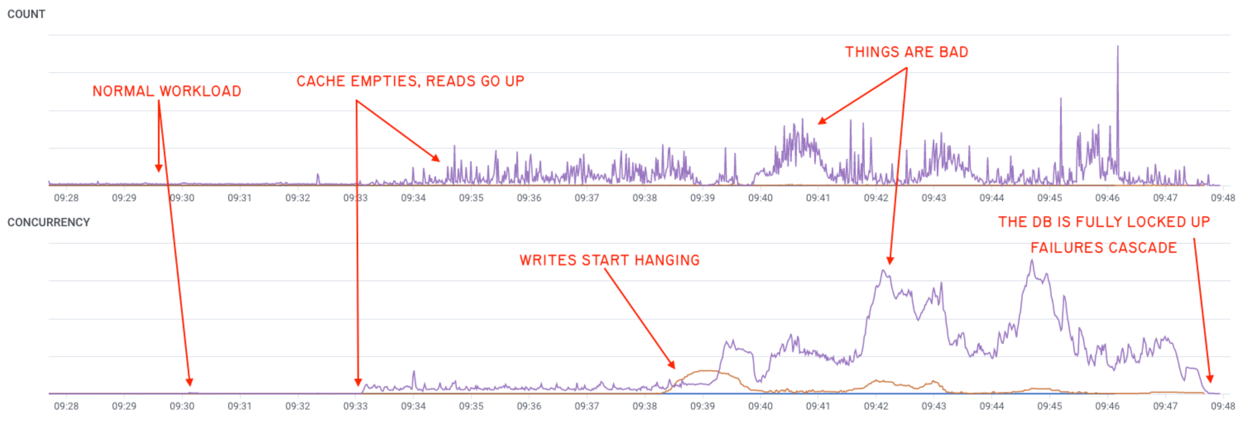
Analysis

As the dust settled and we ran our incident investigation and review, multiple interesting threads came up. One was to figure out how our database died so hard, which naturally led to a discussion about its central role in our infrastructure. We also had a long discussion on how the response felt, once we were stuck in it. Finally, we had a lot of talks about what exactly we can do for this type of outage.

How a database dies

The precise database failure mode we encountered is difficult to explain. After digging into all sorts of metrics and logs, we managed to figure out that at the core of the database failure was a deadlock in MySQL's internals, and not in our transactions.

There's no clear path on how we specifically got to that deadlock, but we think that since this is a concurrency issue, it's non-deterministic, and the chances of hitting these increases with the amount of contention and parallelism going on. The more loaded the database is, the higher the likelihood of hitting these rare bugs. Here's what we observed:



As our cache mostly stopped working, the amount of reads (in purple) we ran shot up drastically, overloaded the database, and then a few writes (in orange)—a decent amount, but nothing out of the ordinary—seemingly tipped it over into a rare race condition in the MySQL internals, locking thread after thread of the database, until operations piled up.

Eventually, all working threads were exhausted, the database ran out of connections, and our services died when they were unable to get the information they needed to run. This internal deadlock further explains why we weren't able to free up connections: anything we did that tried to interact with any stuck thread got stuck waiting on the same lock as well.

Part of the heavy write workload comes from sequences of insertions into our schema storage when new fields are encountered by customers. This is generally a bit demanding for the database, but the heavy read load moved us into a rather dangerous and uncomfortable situation. Engineers described this as "playing golf in a storm," denoting the probabilistic nature of the issue, but also how our system was at the same time put in a vulnerable position.

Our cache implementation is perceived as being load-bearing and structural: the system isn't safe without it. This is known to be a less than ideal situation, particularly when this database is so central to the system, which brings us to our next point on single points of failure.

Single points of failure and ouroboroses

We run half a dozen different MySQL databases, all hosted via RDS. Although we have many, we do have a “main” database that contains many types of data, required for everything related to user accounts, permissions, configuration values around teams, SLOs and triggers, and finally, environment and data schemas—to name the main ones. None of these contain actual customer events, which are stored via our query engine, but they are necessary for the overall system to work.

Most features are directly tied to environments and data schemas, and so this database naturally “attracts” more data to it, as all services tend to require at least some of that information to function. This represents an obvious Single Point of Failure (SPOF), which everyone in the organization is aware of and continuously working on; the other databases we have were mostly split away from the main one over time as an attempt to remove performance bottlenecks and reduce the blast radius for certain features.

Our cache protects this database, and the ingestion of data is required to keep the cache as warm as possible. Cold starts are challenging but manageable; it’s running hot without the cache that causes issues.

As an additional factor, the cache is shared by multiple services, which all keep an in-memory cache and refresh it from the layered one, and fall through to the database when there’s an issue. To maintain correctness, a single process backfills the cache from the database; if the layered cache isn’t around, processes can still update their own, and in theory, the system can run for a long time, as long as the database does not hard lock—which is sadly what happened.

This creates a system where many dependencies are indirect, and things going wrong in one end of the environment can end up degrading service for another. Many of these are set up because they are more economical and less risky.

In fact, that complexity is hardly avoidable. Locally good decisions often interact in unexpected ways:

- The query engine updated the timestamps to the database because it already had to see all the data, and it ensured consistency between what the interface reports and what the queries return.

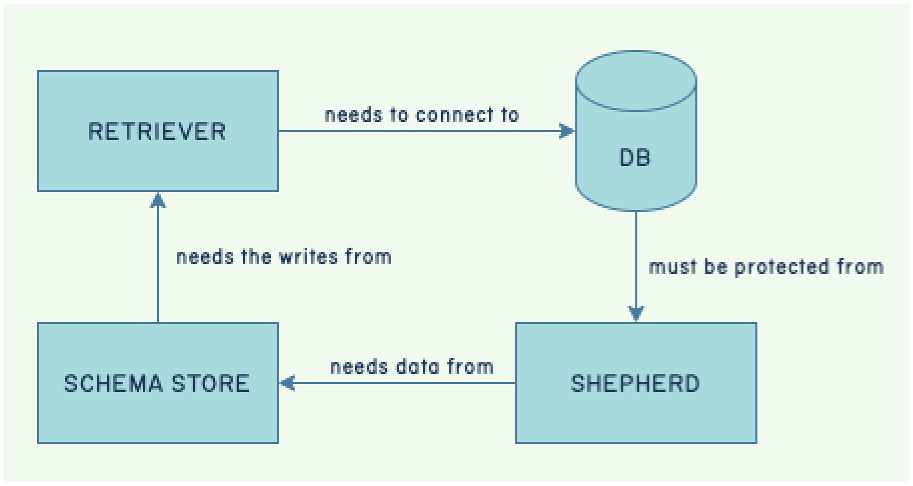
- The shared cache implementation was in part scaling work, and in part an effort to clean up and normalize all schema usage—a first step in making it more manageable to untangle the “main” database usage.
- We believed the failover of the primary database instance to be costly because we had seen inadequate performance issues on replicas when failing over, for many hours that follow.
- We now know that the replicas having reliability and performance issues were a disjoint failure mode, but these prior problems are what made us want to switch the writes across query engine clusters during migrations, and ultimately fed into this outage.
- We were migrating back to avoid a bug with the new cluster version that we thought could corrupt data.
- Frequent deploys mitigated a non-obvious bug around a specific feature flag which everyone therefore believed to be safe.
- During the early investigation, we had paused deployments to prevent interference with new code and host restarts.

As it turns out, preventing a bug by switching to the old infrastructure set up the stage. This investigation made us suspend otherwise frequent deployments, which made it a guarantee that the restarts accidentally required for the query engine flags wouldn’t happen by accident. Finally, recovery on the database was a bit slower because prior near-misses with our databases led us to believe performance could be bad for a longer time than what we observed in practice.

It is a bit ironic how feature flags, frequent deploys, suspending deploys during incidents, and learning from prior near-misses all technically contributed to this incident, while being some of the most trusted practices we have to make our system safer.

Incident response is personal

We made a specific call during the incident to go down hard, only so that we could come back up faster. What we knew at the time was that the load on the database without a cache was the likely problem, that the cache required Retriever to be rebooted to work, and that we couldn’t reboot Retriever until the database was back up, which wouldn’t be safe to assume unless ingest was taken out of rotation.



Under that pattern, the assumption was that any partial failure where we brought ingest back up as fast as we could would not actually get us out of the incident. When this sort of self-sustaining situation happens, the most effective course of action is often to interrupt it entirely to restart it in a better state.

This strategy felt safer, faster, and also more straightforward at the time because of recent experience with incidents, such as issues with our database replicas (to get more familiar around database debugging and failovers), a full Shepherd outage (to exercise the circuit breaking), and experiments with booting fresh clusters of Honeycomb in non-production environments (which demonstrated we don't have circular dependencies that prevent us from cold restarting).

All these fragments of past responses were put together to handle this one bigger incident. Engineers mentioned during the review that the response felt effective and well organized internally, especially when compared to smaller outages in the past.

The moment the team investigating the issue noticed everything turned into an outage, the whole response reorganized in a matter of minutes. It went from a brainstorm-centered approach, where people looked at various things and suggested possible approaches, to one that was a lot more focused on action.

Because responders knew everyone involved and their respective skill sets:

- We self-organized following known patterns for the overall response
- We identified an engineer who had done the ingest circuit breaking and was around to do it as fast as possible to break off the crash loop

- Another engineer self-assigned the role of looking into the database issues and connection problems
- We brought in more engineers who understood MySQL really well to work on that issue when it surfaced
- We started multi-pronged side investigations into how to most effectively backfill the cache, hedging our bets since not all approaches were guaranteed to work

Since they had also dealt with a lot of incidents or high-stakes near-misses together in the past, participants already knew to self-report their work. The self-appointed incident commander was aware of their own tendency to veer off into debugging and monitoring work and both took measures to stay focused and had coworkers reminding them of it, which opened the door to side investigations being kept in the loop more effectively.

Finally, people whose tasks were under control passively listened to ongoing investigations (e.g., idling in Zoom calls) while our engineer handling comms made frequent updates in shared channels. Everyone already had context into the incident. When all the pieces were in place, we could join forces to merge the cache backfill, circuit breaking reopening, and database health monitoring into a single operation.

In the end, the incident was high pace, but it didn't feel confusing. We had a good balance of orchestration and self-organization.

Corrective actions

For the duration of the outage, no new data could be ingested. If our users or customers did not buffer this data to disk somewhere to replay it later, it is gone for good. This has knock-on effects during the incident on alerting and querying, but the ingestion gap stays for as long as there's data retention. As such, major ingest outages like that have a way to stretch in time beyond the period during which the system is down.

This type of failure mode is something both we and our customers understand is possible based on how we are only present in one region (although we do run in multiple redundant zones). The presence of SPOFs, while possibly less visible to people outside of Honeycomb, is well known—and a frequent concern internally. Migrations away from such patterns are considered safer when done gradually. Moving fast requires stopping all activity, even urgent scaling work, and rushing can cause even more major outages.

The consequence of this is that nobody has seriously suggested "blow up the database and remove the single point of failure" as an outcome of this incident, because it's both impractical as an action item, and also something that is always on people's minds and gradually being worked on when doing their engineering job. Breaking apart SPOFs is part of a less explicit set of long term objectives that we bake into designs, but having this sort of trade-off be made more explicit in the aftermath of an incident review is something we are discussing within engineering teams.

We're exploring future architectures that may mitigate the impact of such incidents and their implications. We are looking at ways to strengthen the cache further, mechanisms to lower the amount of contention we put on our database during "update" storms from new schemas, and to stabilize the performance costs of some operations within our system. We are studying them along other measures, including those related to instrumentation or experimentation to better detect and handle such edge cases.

In the short term, the chances of an incident happening with this specific failure mode have been drastically reduced, mostly because our migration is complete and we have already removed all code that allows writes behind the cache to be disabled. The expected response time is also likely to be much faster since we've updated our understanding of database failures and make recovery faster for unlikely lockups of this kind. In the meantime, we are carefully evaluating the options above to see which represent the best trade-offs.

Conclusion

This incident is something we wish wouldn't happen, but know that from time to time, we'll have to manage them—no matter how hard we plan for them. One of the most intriguing themes that came up from many of our recent incidents and near-misses is how often one of their contributing factors has been trying to prevent a subtle bug with possibly bad consequences from happening—only to end up with a bigger unforeseen outage at the end.

The process around incidents is generally paved with good ideas for improvements, the best of intentions, and fascinating surprises. We hope that we've been able to illustrate how these came to interact with each other in this specific situation, and that they convey the complexity and richness behind this outage.

Some of the patterns in incident response mentioned here may feel familiar with some of our readers. If you've seen similar things and want to discuss them with us, reach out to us in Pollinators, our Slack community.