

ELLYPUNK

Advanced

WEEKLY DIGEST / A301

28 September - 2 October 2026



This week

28 September - 2 October 2026

Select a title to turn to its pages, or open the original online.

MONDAY / 28 September 2026

Docker Memory Limits: Diagnose an OOM Kill

3

[Open original online](#)

TUESDAY / 29 September 2026

Docker Network Namespaces: Trace Container-to-Container Traffic

8

[Open original online](#)

WEDNESDAY / 30 September 2026

SQL Sort Spills: Diagnose a Query That Runs Out of Memory

13

[Open original online](#)

THURSDAY / 1 October 2026

SQL Cardinality Estimates: Compare Predicted and Actual Rows

18

[Open original online](#)

FRIDAY / 2 October 2026

Docker Build Cache: Diagnose Unnecessary Layer Rebuilds

23

[Open original online](#)

Original artwork is reproduced in full, in publication order.

Copyright Ellypunk. Contains AI-generated artwork.

ellypunk.com



MEMORY LIMITS

DIAGNOSE AN OOM KILL

WORKLOAD
PEAK

CONTAINER
LIMIT

KERNEL
OOM

An OOM kill happens when the workload exceeds its available memory under pressure.

Example values are illustrative.

1. CONFIGURED CAP

First, check the container's memory limit. This is the most the container is allowed to use.



CONFIGURED CAP

Memory limit:
512 MiB

2. CURRENT USAGE

Then look at the current memory usage. If it approaches or exceeds the limit, it's a risk.



CURRENT USAGE

480 MiB / 512 MiB

94%

3. HOST PRESSURE

Also check the host's overall memory pressure. If the host is low on memory, the kernel is more likely to kill containers.



HOST PRESSURE

Host memory usage:

92%

An OOM kill usually means the workload's peak usage exceeded the container's memory limit while the host was under memory pressure.

THE BIG PICTURE

WORKLOAD
PEAK
600 MiB

CONTAINER
LIMIT
512 MiB

HOST
PRESSURE
High
(92%)

Peak exceeds limit

Host under pressure

KERNEL
OOM

Container is killed





BUILD THE EVIDENCE

Don't rely on one signal.
Corroborate evidence from multiple sources.

Triangulate multiple sources to find the real cause.

ILLUSTRATIVE TIMELINE

10:31:08 • DOCKER OOM EVENT

Docker emits an oom event for container "api".

EVENTS

10:31:08 • OOMKilled = true

Container state shows OOMKilled = true at the same timestamp.

STATE

10:31:09 • EXIT CODE 137

Container exits with code 137 (one second later).

EXIT

Exit 137 alone shows SIGKILL, not its cause.

Corroborate with a Docker OOM event and the container state at the same time.

THE COMMANDS

Use these three commands to collect the evidence.

1 Get Docker OOM events

```
docker events --filter 'container=api' --filter 'event=oom'
```

2 Check container state

```
docker inspect -f '{{.State.OOMKilled}} {{.State.ExitCode}}' api
```

3 Check current resource usage

```
docker stats --no-stream api
```



TIP

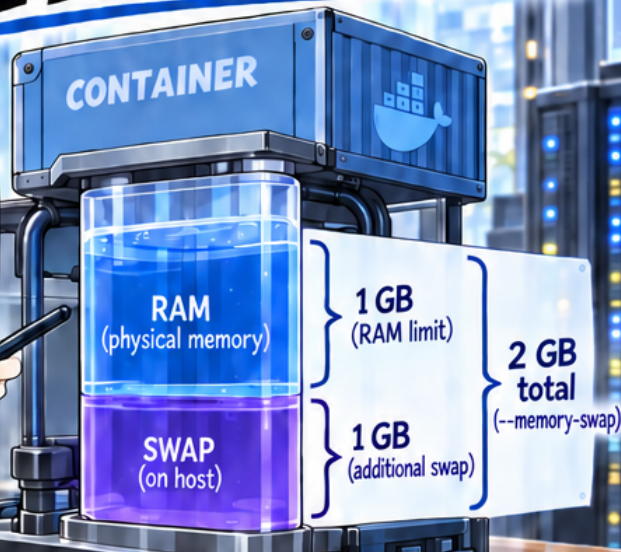
Match the container and timestamp across all three checks.



READ THE LIMITS



Docker gives you different controls for memory. They work together, but they mean different things.



1 --memory=1g

Hard RAM limit: 1 GB

The container cannot use more than 1 GB of RAM. If it tries to go higher, it will be OOM killed (by the kernel) inside the container.



2 --memory-reservation=768m

Soft limit: applies during memory contention; keep it below --memory

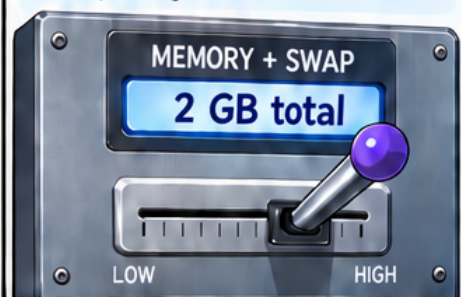
This is a soft target of 768 MB. The container can use more than this when there is free memory, but it may be reclaimed first if the host is under memory pressure. Set it lower than --memory.



3 --memory-swap=2g + --memory=1g

2 GB total: 1 GB RAM + up to 1 GB swap

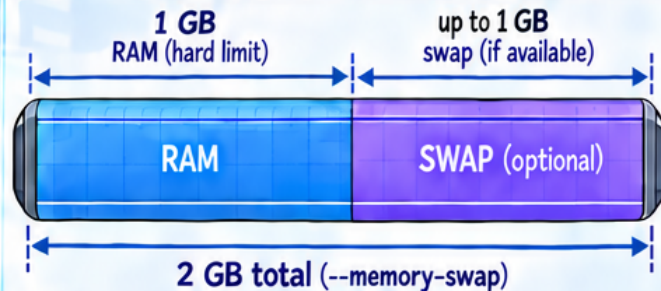
With --memory=1g and --memory-swap=2g, the container can use 1 GB of RAM and up to 1 GB of swap (2 GB total). Swap use depends on the host's swap configuration.



Reservation is a soft target, not a second hard cap.

⚠ Swap use depends on host configuration.

EXAMPLE ALLOCATION (WITH THESE OPTIONS)





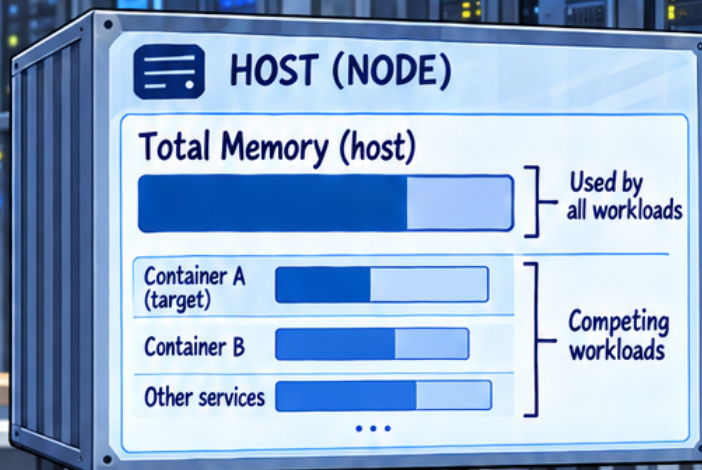
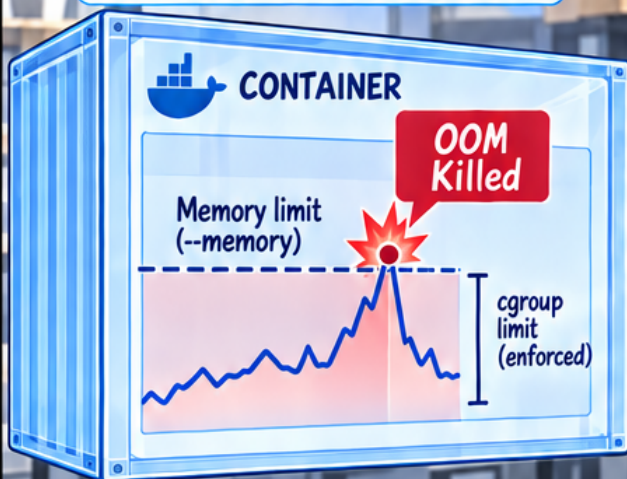
LOCATE THE PRESSURE

CONTAINER LIMIT

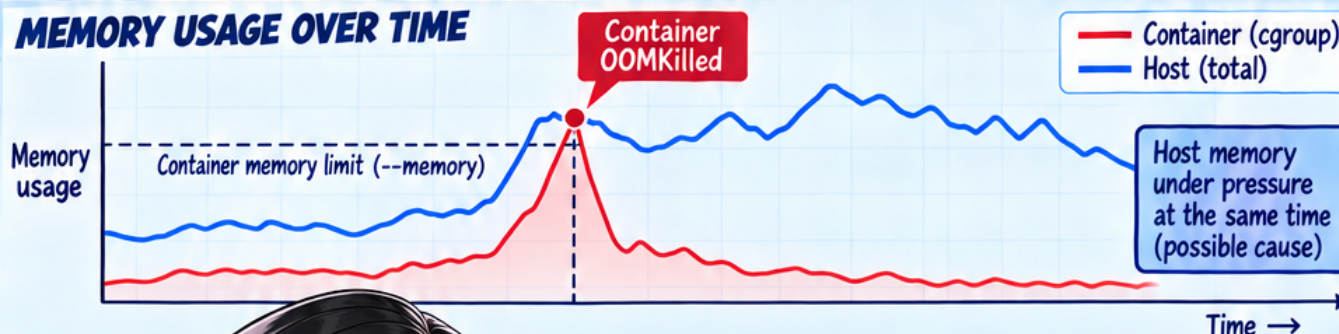
- 1 OOMKilled = true
- 2 Compare workload peak with --memory.

HOST PRESSURE

- 1 Check total host memory and competing workloads.
- 2 Correlate the kernel event time.



MEMORY USAGE OVER TIME



docker stats is a current snapshot; it can miss a short peak before the kill.



Was the cap crossed, or was the host short of memory?



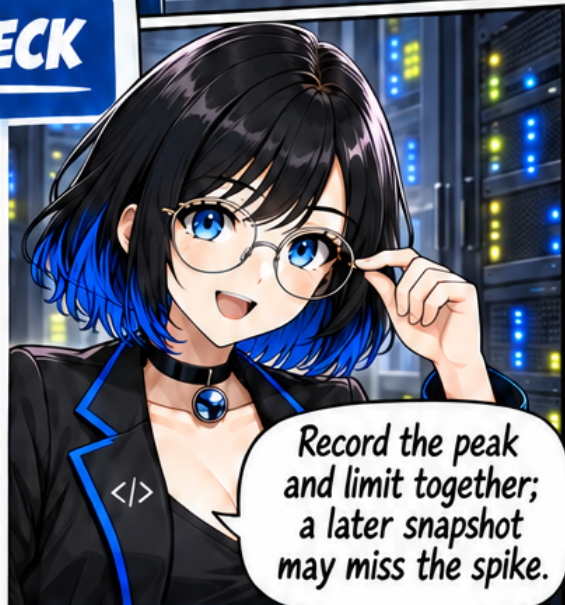


THE POST-INCIDENT CHECK



THE POST-INCIDENT CHECK

- 1 Pinpoint the container and incident time.
- 2 Correlate docker events with .State.OOMKilled and the exit code.
- 3 Compare --memory, --memory-reservation and --memory-swap.
- 4 Measure peak use under representative load and leave host headroom.
- 5 Repeat a controlled load test and check for a new OOM event.



Do not disable OOM killing without a memory limit.





DOCKER NETWORK NAMESPACES: TRACE CONTAINER TRAFFIC

OBJECTIVE:

Follow a request from one container to another on a user-defined bridge.

CONTAINERS
NETWORKING
OBSERVABILITY

EACH CONTAINER

Each has its own network namespace.

web



Network
Namespace

eth0

db



Network
Namespace

eth0

SHARED BRIDGE

A shared user-defined bridge and DNS connect peers by name.

web



db



app-net (bridge)

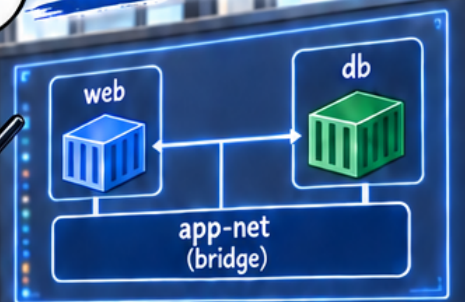


DNS



TRACE THE PATH

Let's trace the path from web to db.



CONTAINERS
NETWORKS
NAMESPACES
PACKET FLOW
DNS
BRIDGE

1 • web resolves db

User-defined bridge DNS returns db's address.



2 • web sends

The packet leaves web's network namespace.



3 • bridge forwards

A virtual link carries traffic across the shared bridge.

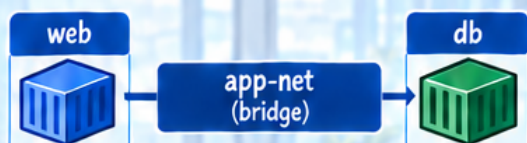


4 • db receives

The packet enters db's network namespace.



Both containers must join the same intended network.



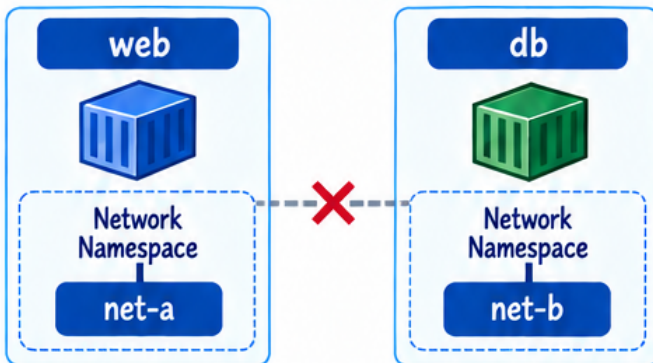


TWO DIFFERENT FAILURES



NAME FAILS

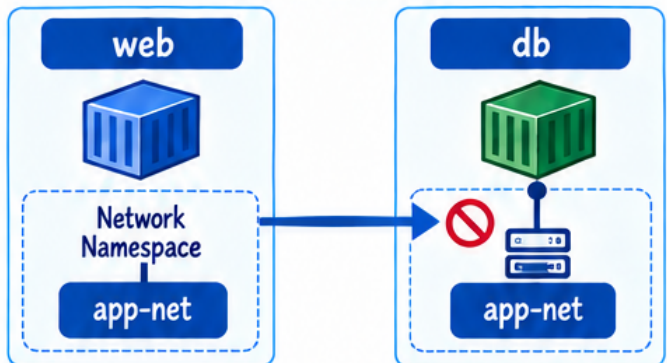
web cannot resolve db.



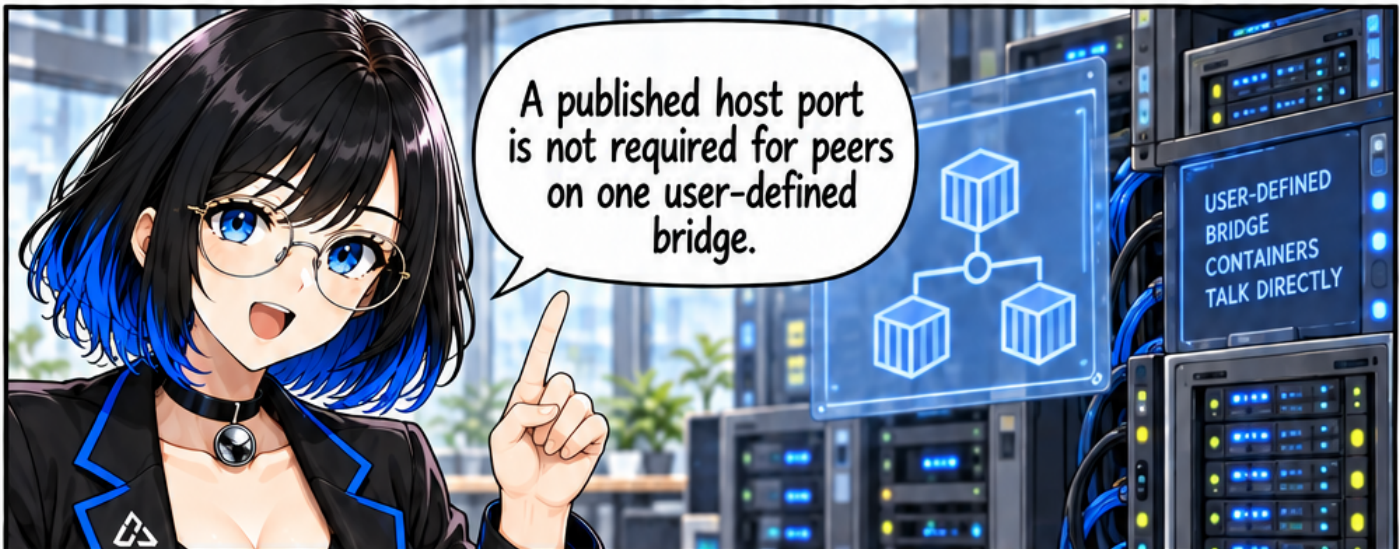
Check shared network membership.

NAME WORKS, PORT FAILS

db resolves, but the service is not reachable.



Check the listening address and port.





db isn't reachable yet?
It's not on the same network!

WORKED EXAMPLE:

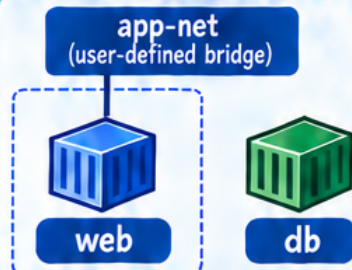
RESTORE SHARED MEMBERSHIP

CONTAINERS
NETWORKING
OBSERVABILITY

1 PROBLEM

web can resolve its peers, but not db.

web is on app-net,
but db is not,
so they can't
reach each other.



2 CHECK

docker network inspect app-net

Let's inspect
the network to
see which containers
are attached.



Docker CLI • Linux / macOS / Windows

docker network inspect app-net

The Containers list shows **web**, not **db**.

3 ACTION

docker network connect app-net db

Attach db to
the same network.

Docker CLI • Linux / macOS / Windows

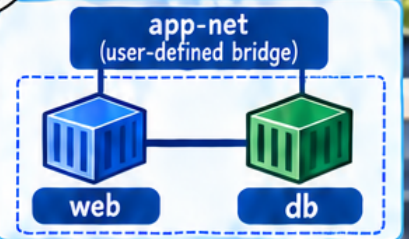
docker network connect app-net db

Only if network policy permits this connection.

4 VERIFY

docker network inspect app-net

Now both
containers are on
app-net. Test the db
service name and
listener from web.



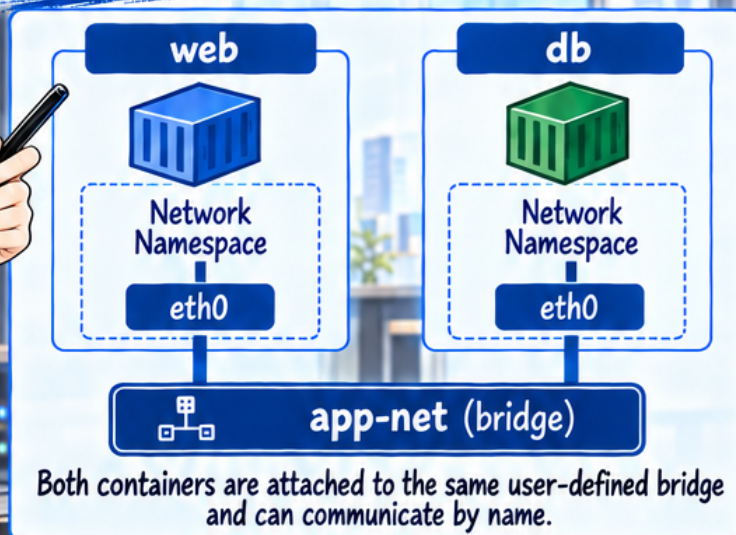
Docker CLI • Linux / macOS / Windows

docker network inspect app-net

Both **web** and **db** now appear. Test the **db** service name and listener from web.

Let's trace container-to-container traffic!

TRACE CONTAINER TRAFFIC



RECAP

Three key points to remember!

- 1 Each container has a separate network namespace.
- 2 Peers on one user-defined bridge can resolve container names.
- 3 A published host port is not needed for bridge peers.

CHECKLIST

- ☐ Confirm web and db share app-net.
- ☐ Resolve db by name from web.
- ☐ Check the db service is listening on the expected address and port.

CAUTION

On Docker Desktop for macOS and Windows, the bridge lives inside a VM. Use Docker CLI inspection; do not expect docker0 on the host.

ELLY'S FINAL TIP

If name lookup works but the connection fails, inspect the service listener before changing network membership.

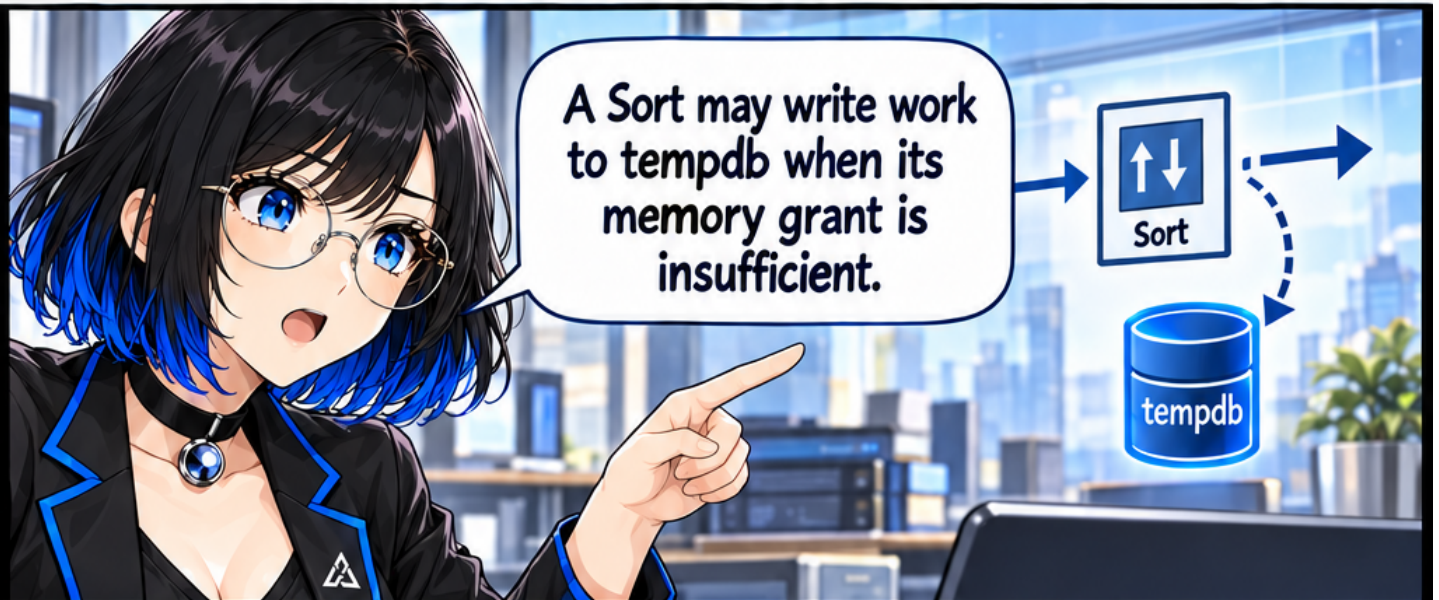
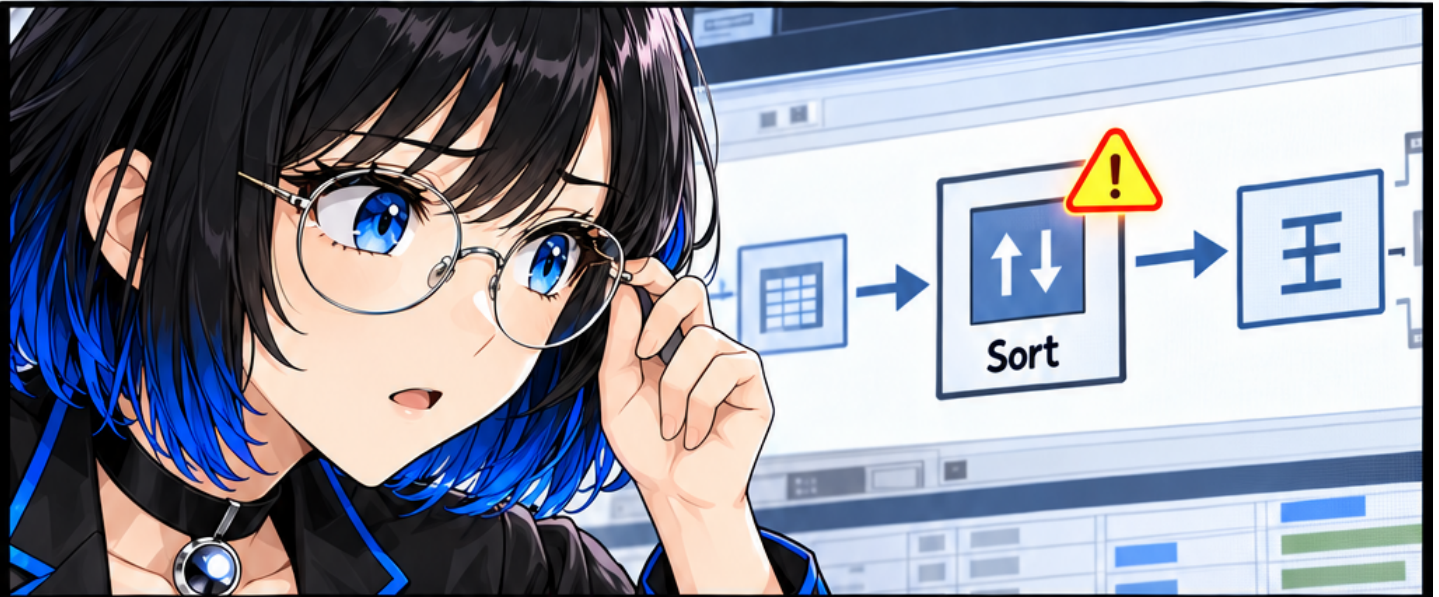


SQL SORT SPILLS

Diagnose a query that runs out of memory.

OBJECTIVE:

Find the spill warning in the actual execution plan.



READ THE ACTUAL PLAN

1

1. Find the Sort spill warning.



2

2. A Sort can spill work to tempdb.



3

3. Compare estimates and memory grant.



A warning is evidence, not a diagnosis.

Find the cause

Check row estimates, input width, and concurrent memory pressure.

An index may supply the needed order and avoid a Sort. Verify the new plan and write cost.



A spill warning alone does not identify the cause.

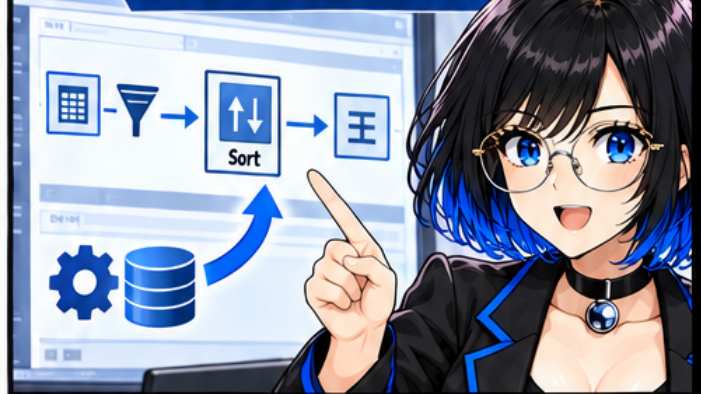
Test one change



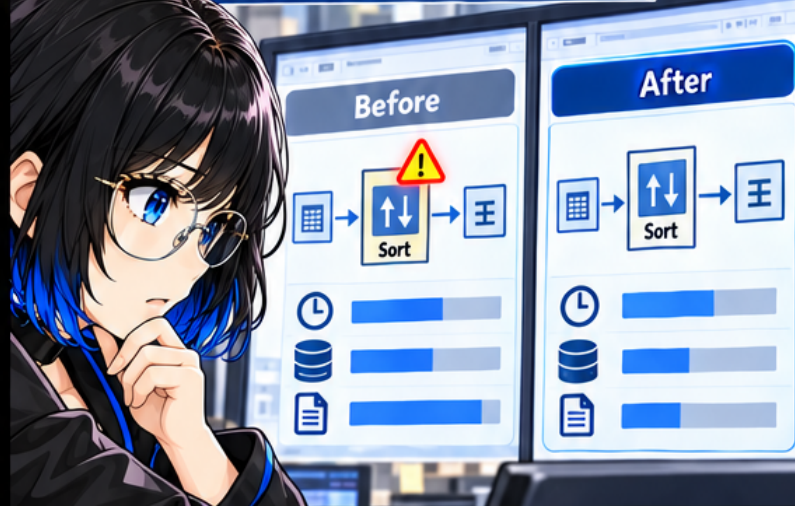
1 Problem: an ORDER BY query shows a Sort spill warning.



2 Action: check estimates, then test an index that matches the filter and order.



3 Check: compare actual plans, spill warning, elapsed time, and writes.



4 Improvement: keep the index only if the full workload benefits.



SPILL DIAGNOSIS CHECKLIST

1. Capture the actual plan.



2. Confirm which Sort spilled.



3. Review row estimates and memory grant.



4. Test one change and compare the workload.



Final Tip

Do not add an index solely because a Sort spilled; measure reads, writes and elapsed time.



SQL CARDINALITY ESTIMATES

COMPARE PREDICTED AND ACTUAL ROWS.

An estimate is the optimizer's predicted row count; actual rows come from execution.

OBJECTIVE:

Find the plan operator where the estimate diverges before changing the query.

A SIMPLE EXECUTION PLAN OPERATOR



Hash Match
(Inner Join)

Estimated rows: 100

Actual rows: 100,000

ESTIMATE DIVERGES HERE.

1. CHECK ESTIMATED VS ACTUAL ROWS

Compare the estimated rows and the actual rows on each operator.

Look for the operator where the numbers diverge significantly before changing the query.

2. UNDERSTAND THE NUMBERS

Estimated rows

The optimizer's predicted row count for the operator.

Actual rows

The number of rows that really flowed through the operator during execution.

READ THE ACTUAL PLAN

Let's read the execution plan to see how the estimate compares to what actually happened.

1

Estimated plan:

Shows the optimizer's predicted number of rows. No execution counts.

2

Actual execution plan:

Contains runtime actual rows from when the query ran.

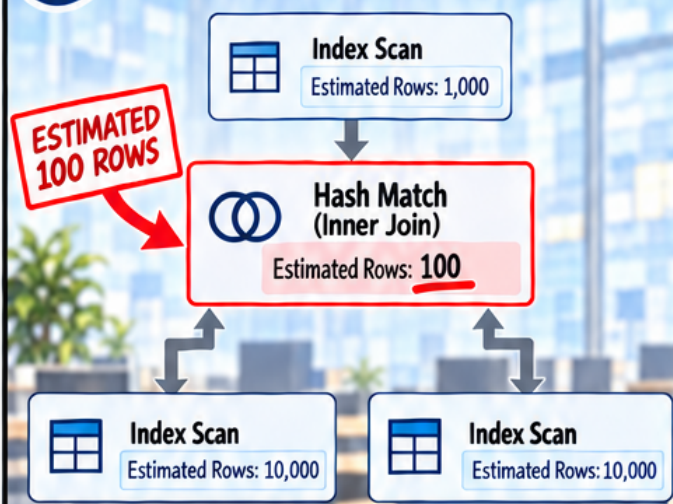
3

At the same operator:

Compare the Estimated Number of Rows with the Actual Number of Rows.

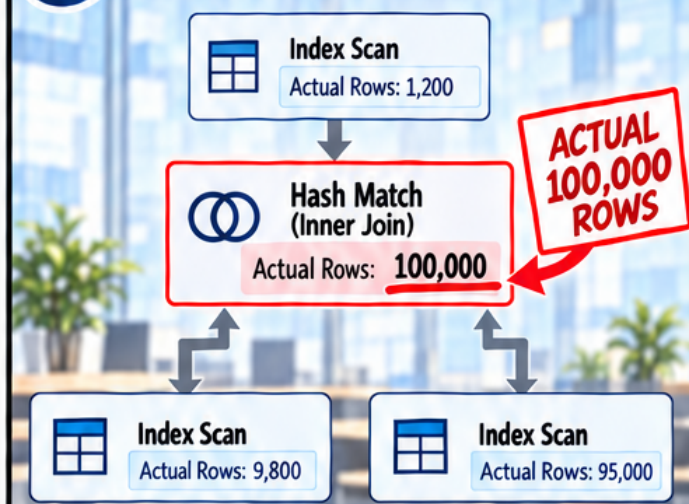
1

ESTIMATED EXECUTION PLAN



2

ACTUAL EXECUTION PLAN



At the same operator, the estimated rows are 100, but the actual rows are 100,000. That's a big difference, and it can lead to a suboptimal plan choice.



IMPORTANT

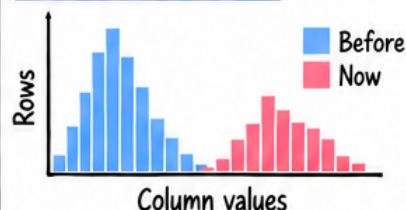
Run a representative query and inspect its actual plan; workload and parameters matter. Different data and parameters can lead to very different results.



ASK WHY THE ESTIMATE MISSED

A mismatch is evidence to investigate, not proof of stale statistics.

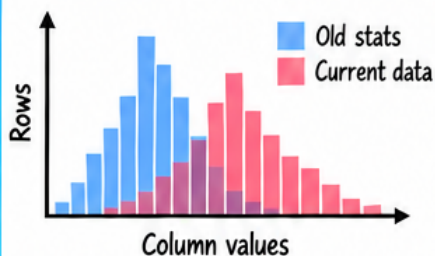
Data Distribution



THREE PLACES TO INVESTIGATE

1 STATISTICS

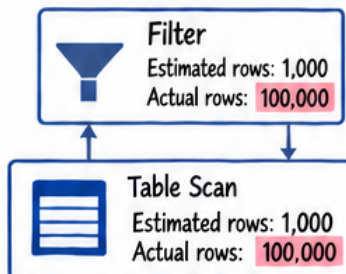
Has the data distribution changed since the last update?



If the distribution, density, or number of distinct values has changed, the old statistics may no longer be accurate.

2 PREDICATE

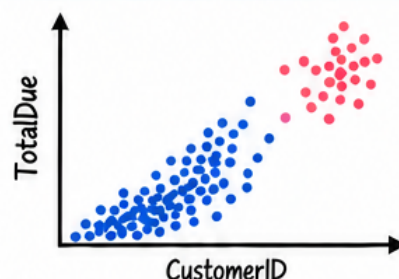
Does a function or implicit conversion hide useful column information?



Functions or implicit data type conversions can prevent the optimizer from using column statistics effectively.

3 CORRELATION/SKEW

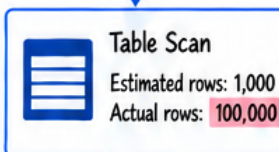
Are values uneven or dependent?



Values can be skewed or correlated with other columns. The optimizer may assume independence and misestimate the number of rows.



ACTUAL vs ESTIMATED



Large difference means investigate the causes.

The estimate is way off. Let's diagnose why, fix it properly, and confirm the result.

WORKED DIAGNOSIS

Find the cause.
Apply a targeted fix.



1 PROBLEM: The operator estimated 100 rows; actual plan returned 100,000.



Hash Match
(Inner Join)

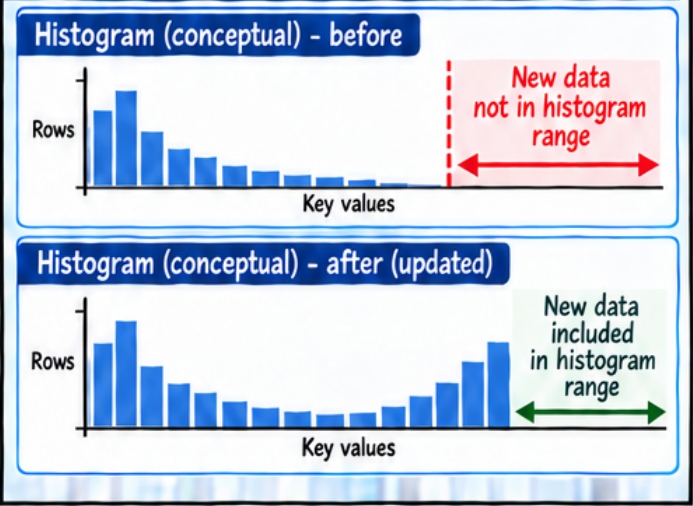
Estimated rows: 100

Actual rows: 100,000



The estimate is 1,000× lower than what actually happened. Something is missing from the optimizer's view of the data.

2 CHECK: Recently loaded data is missing from the histogram range.



3 ACTION: Review the relevant statistics and update them only if justified.



Check when the stats were last updated and whether they reflect the recently loaded data. Update statistics if they are out of date and it's appropriate for your workload.

- Review the relevant statistics (e.g. row count, last updated time).
- Update only if justified by the data characteristics.
- Do not rely on automatic blanket updates.

4 VERIFY: Rerun a representative query; compare the same operator and elapsed work.



Hash Match
(Inner Join)

	Before	After
Estimated rows	100	98,000
Actual rows	100,000	100,000
Elapsed time	2.8 s	2.6 s



The estimate is now in line with reality, and the plan is more appropriate. Always test under a normal workload to confirm the overall impact.



A changed estimate can change the plan; test under normal workload.

Let's go through a practical checklist!

CARDINALITY CHECKLIST

COMPARE THE SAME OPERATOR.

These checks help you systematically investigate why estimated and actual rows diverge.

1 CAPTURE A REPRESENTATIVE ACTUAL EXECUTION PLAN.



Capture a representative actual execution plan.

2 AT ONE OPERATOR, COMPARE ESTIMATED WITH ACTUAL ROWS.



Estimated rows:	100
Actual rows:	100,000

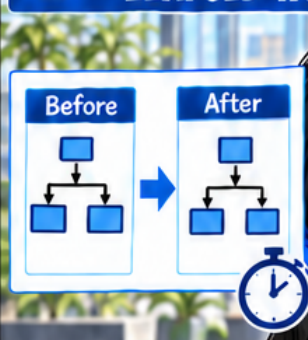
At one operator, compare estimated with actual rows.

3 INVESTIGATE DATA DISTRIBUTION, PREDICATE SHAPE AND STATISTICS USED.



Investigate data distribution, predicate shape and statistics used.

4 AFTER A JUSTIFIED CHANGE, RERUN AND COMPARE PLAN PLUS ELAPSED WORK.



After a justified change, rerun and compare plan plus elapsed work.

RECAP



A large mismatch points to a question; it does not name the cause.

FINAL TIP

Record the operator and both row counts before changing statistics.





DOCKER BUILD CACHE:

FIND THE INVALIDATED LAYER.

OBJECTIVE:

Trace the first cache miss, then move stable steps earlier.

CONTAINERS

DEVELOPER EXPERIENCE

SCALABLE SYSTEMS

CLEAN BUILDS

1. DOCKERFILE LAYER ORDER

A Dockerfile is built in ordered layers. Each layer is cached based on its content.



2. A SOURCE CODE CHANGE

When you change application source code, only the layers that depend on those files are invalidated.

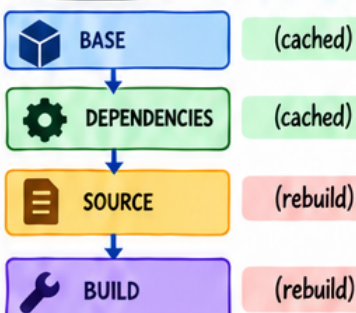
src/app.py

```

10 ...
11 - return "Hello"
12 + return "Hello, world!"
13 ...
    
```

3. WHICH LAYERS REBUILD?

A change in the source invalidates the SOURCE layer and all following layers, but earlier layers remain cached.



4. RESULT: FOCUSED REBUILD

The first cache miss is the SOURCE layer. So we rebuild SOURCE and everything after it, while BASE and DEPENDENCIES are reused from cache.

KEY TAKEAWAY

Trace the first cache miss, then move stable steps earlier to keep rebuilds small.



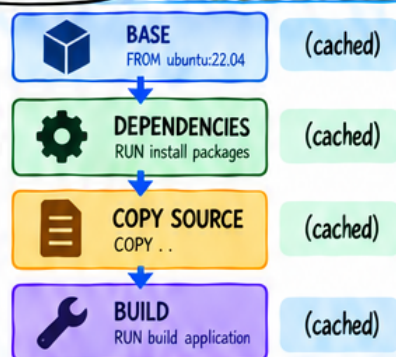
WHY THE CACHE MISSES

When the first COPY input changes, Docker marks that layer as invalid, and everything after it must be rebuilt.

A layer's cache key depends on its inputs. Change the inputs to COPY, and this layer misses the cache. All subsequent instructions run again because their parent layer changed.

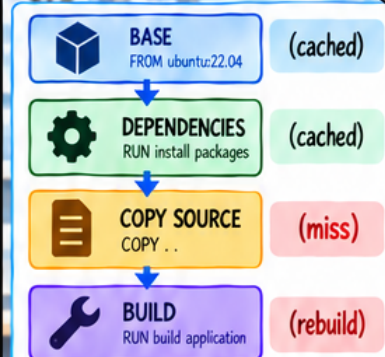
1 LAYER FLOW (INITIAL BUILD)

On the first build, each instruction creates a layer that is cached based on its inputs.



2 SOURCE CHANGES

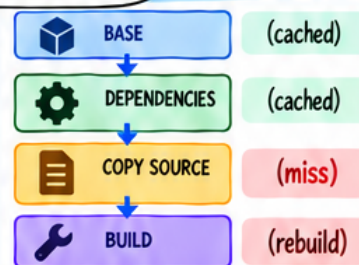
You change a file that is part of the COPY input, so this layer can no longer be reused.



3 EVERYTHING AFTER REBUILDS

Because the COPY layer is different, its digest changes. All following instructions depend on it, so they are executed again.

Earlier layers that do not depend on the changed files remain cached.



4 FILE CONTENT VS. METADATA

Docker compares the file content and relevant metadata (path, size, permissions, etc.) for COPY. A change in the file's modification time alone does not invalidate the cache.

Content changed (e.g. different bytes, permissions, or path)	Invalidates COPY cache
Only mtime changed (timestamp updated, same content and metadata)	Does not invalidate

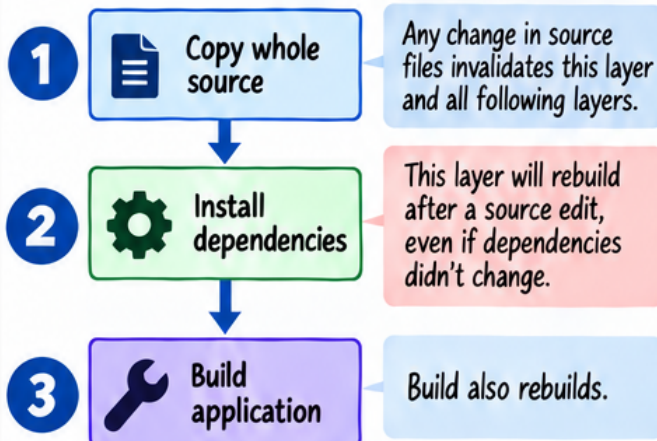
ORDER STABLE INPUTS FIRST

Docker reuses a layer only when the instruction and its inputs are unchanged. So the order of instructions matters.

Put the inputs that change frequently as late as possible.

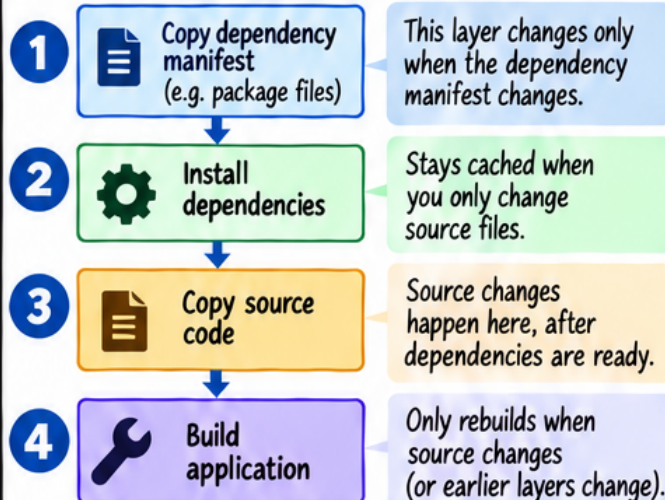
BEFORE – SOURCE FIRST

Copying the whole source early causes the dependency step to rebuild when you edit any source file.



AFTER – STABLE INPUTS FIRST

Copy only the dependency manifest first, so the dependency layer can be cached across source changes.



By ordering the Dockerfile from stable inputs to frequent changes, you let Docker reuse more layers and keep rebuilds fast.

! IMPORTANT CAVEAT

Changing the dependency manifest will still rebuild the dependency layer, and all following layers.

That's expected — the dependencies really did change, so they need to be reinstalled.



WORKED EXAMPLE: FIND THE FIRST MISS

PROBLEM:

Editing `app.py` unexpectedly reruns dependency install.

Editing a small source file shouldn't reinstall all dependencies!
Let's find out why.

1 OBSERVE: THE INITIAL ORDER

The current Dockerfile copies the whole source early, so a source change invalidates multiple layers.

When `app.py` changes, the `COPY` layer changes, which invalidates all following layers.

BEFORE (CURRENT ORDER)

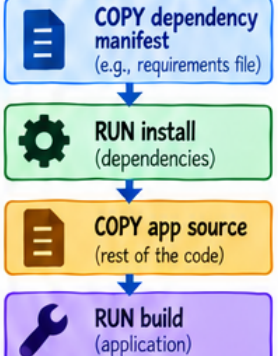


2 ACTION: REORDER THE LAYERS

Copy only the dependency manifest first, then install, then copy the rest of the source.

Separate what changes often from what rarely changes.

AFTER (BETTER ORDER)



3 CHECK: SOURCE-ONLY EDIT

Edit `app.py` and rebuild. See which layers run.

With this order, a source-only change should reuse the dependency install layer.

REBUILD RESULT (after editing `app.py`)



4 IMPROVEMENT: FEWER REBUILDS

Reordering keeps dependency installs cached when only source files change.

Now a source change only rebuilds what needs to.

RESULT



Fewer repeated dependency installs.

KEY TAKEAWAY

Copy dependency manifests first, then install. Keep frequently changing source files in later layers to minimize rebuilds.





CACHE DIAGNOSIS CHECKLIST

Find the first cache miss.

1



Find the first cache miss.

2



Ask which input changed.

3



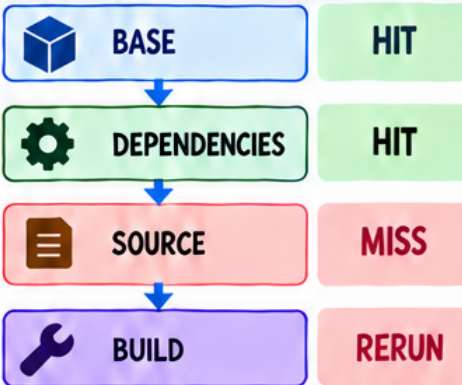
Move stable dependency steps before frequent source changes.

Follow the checks in order to find the real cause.



A changed dependency manifest correctly rebuilds the dependency layer.

RECAP: WHAT HAPPENS



Earlier layers are reused. The change in source invalidates this layer, so everything after it runs again.

Compare two build logs after a source-only edit to prove the install layer was reused.



ELLY'S FINAL TIP

Compare build logs after a source-only edit.

