

ELLYPUNK

Advanced

WEEKLY DIGEST / A301

21-27 September 2026



This week

21-27 September 2026

Available archived editions, in their original date order.

MONDAY / 21 September 2026

SQL Deadlocks: Find the Cycle, Retry the Transaction

3

[Open original online](#)

TUESDAY / 22 September 2026

Cursor Pagination: Keep Results Stable While Data Changes

8

[Open original online](#)

WEDNESDAY / 23 September 2026

Docker Health Checks: Separate Startup from Readiness

13

[Open original online](#)

THURSDAY / 24 September 2026

SQL Isolation Levels: What Each Transaction Can See

18

[Open original online](#)

FRIDAY / 25 September 2026

SQL Indexes: Choose a Key for a Real Query

23

[Open original online](#)

Only retained editions are included; gaps have not been filled.

Copyright Ellypunk. Contains AI-generated artwork.

ellypunk.com



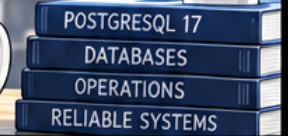
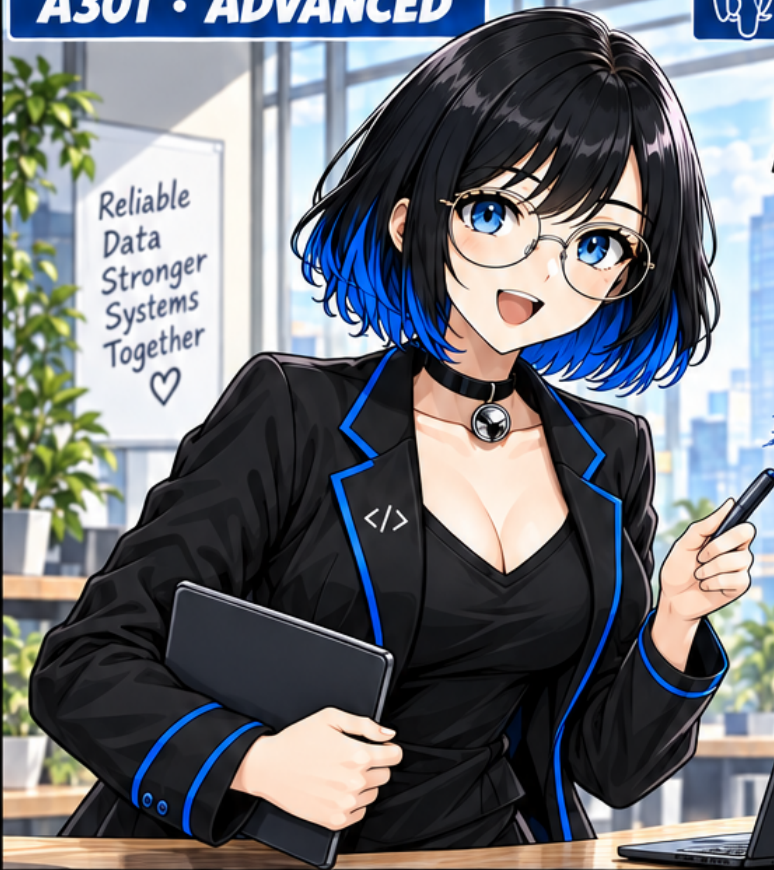
SQL DEADLOCKS

**FIND THE CYCLE.
RETRY THE TRANSACTION.**

OBJECTIVE:

Trace a wait cycle, recover safely, and reduce repeat failures.

OBSERVE
UNDERSTAND
SOLVE
BUILD
BETTER



HOW A LOCK WORKS

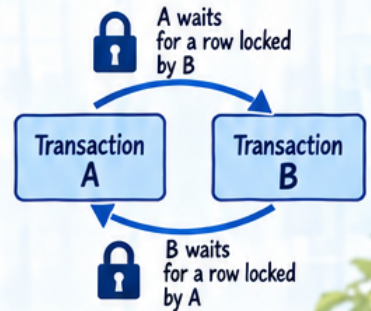
A lock protects a row while a transaction changes it.



Row locked by Transaction A

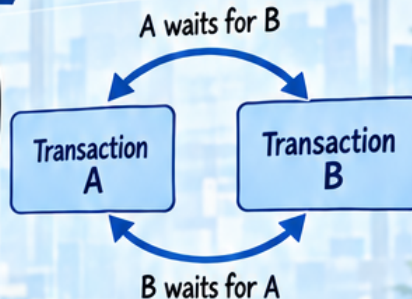
WAITING AND A CYCLE

Waiting is normal. A cycle cannot finish by itself.



A DEADLOCK IS A CYCLE

Follow the waits to find the cycle. Abort one transaction and retry it so the work can complete.



DEADLOCK CYCLE

CANNOT FINISH BY ITSELF.



- ✓ Find the cycle
- ✓ Abort one transaction
- ✓ Retry the transaction
- ✓ Reduce repeat failures

STABLE DATA.
REAL PROGRESS.





1 • Session A locks row 1.

Session A locks row 1.

Session	Row 1	Row 2
Session A	🔒	
Session B		

accounts	
id	balance
1	100
2	200

2 • Session B locks row 2.

Session B locks row 2.

Session	Row 1	Row 2
Session A	🔒	
Session B		🔒

accounts	
id	balance
1	100
2	200

3 • A waits for row 2.

Session A waits for row 2 held by B.

Session	Row 1	Row 2
Session A	🔒	WAIT →
Session B		🔒

accounts	
id	balance
1	100
2	200

4 • B waits for row 1.

Session B waits for row 1 held by A.

Session	Row 1	Row 2
Session A	🔒	
Session B	← WAIT	🔒

accounts	
id	balance
1	100
2	200

A WAITS FOR B ↔ B WAITS FOR A

That closed loop is the deadlock.

DEADLOCK

CIRCULAR WAITS STOP PROGRESS



Real
Systems
Real
Understanding
♡

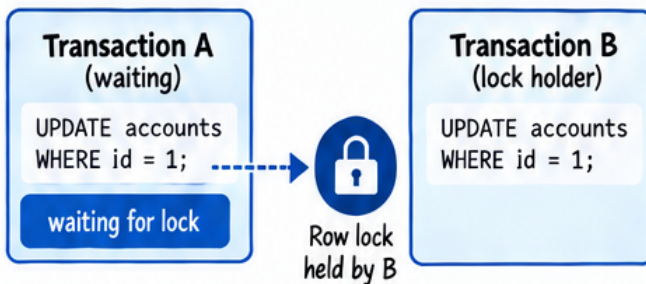
WAIT OR DEADLOCK?

Same symptom – both are waiting.
But they're different. Let's look
at the difference.

Observe
Understand
Solve
Build Better
♡

ORDINARY WAIT

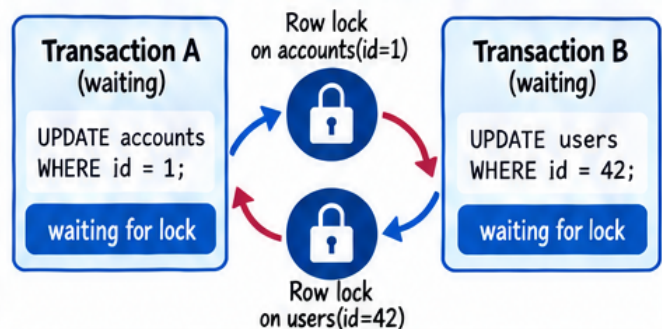
One transaction waits.
The holder can still commit or roll back.



Transaction A waits. Transaction B can
still commit (releasing the lock) or roll back
(releasing the lock). Progress is possible.

DEADLOCK

Each transaction waits for the other.
No participant can progress.



PostgreSQL detects the cycle and aborts
one transaction.

SQLSTATE 40P01 • deadlock_detected

Do not assume which transaction becomes the victim.

WHAT TO DO

Handle deadlocks
in your application
by retrying the
transaction.

- 1 Catch SQLSTATE 40P01 (deadlock_detected).
- 2 Rollback the failed transaction.
- 3 Retry the transaction (with backoff).
- 4 Keep transactions short and access rows in a consistent order to reduce deadlocks.

Detect
Rollback
Retry
Build
Resilient
Systems
♡

POSTGRESQL 17
CONCURRENCY
TRANSACTIONS
BETTER SYSTEMS

WORKED RECOVERY

From deadlock to a safe, reliable result.

Resilient Systems
Serve People
♡

OBSERVE
RECOVER
IMPROVE
TOGETHER
♡

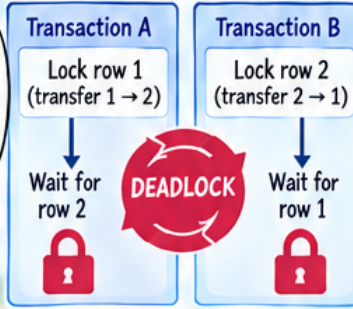
Stability through care
♡

DATABASES
OPERATIONS
BETTER SYSTEMS

1 PROBLEM: The transfer receives SQLSTATE 40P01.



Two transactions try to transfer in opposite directions and lock the same rows in a different order.



The transfer receives SQLSTATE 40P01.

2 ACTION: Roll back and retry the complete transaction.



Use a small retry limit, backoff, and jitter.

- ROLL BACK** (whole transaction)
- RETRY** (limited attempts)
- BACKOFF** (increasing delay)
- JITTER** (randomized delay)

A retry may still fail — handle the error if it does.

3 CHECK: Verify exactly one committed transfer.



After a successful retry, confirm the final state shows exactly one committed transfer.

Final account balances (after recovery)

Account 1 (id = 1)

Account 2 (id = 2)

Exactly one committed transfer. No duplicates.

If the retry also fails, report the error and keep data correct.

4 IMPROVEMENT: Lock account rows in the same order: 1, then 2.



A consistent locking order prevents this deadlock.

Always lock in this order:



Account 1 (id = 1) → Account 2 (id = 2)

Lock order: 1, then 2.

Guardrail: external side effects must be outside the replay—or idempotent.

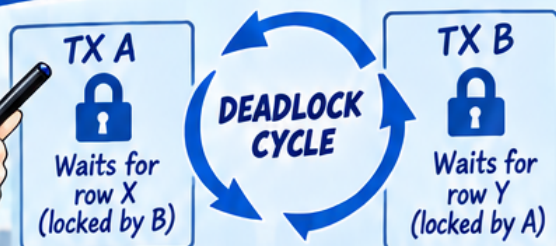




Here's the recap!

SQL DEADLOCKS

FIND THE CYCLE, RETRY THE TRANSACTION



Understand
Detect
Recover
Build
Resilient
Systems

SAME PRINCIPLES. MORE RELIABLE SYSTEMS.



Follow this checklist in production!

FIND
FIX
RETRY
SHIP
CONFIDENTLY

DEADLOCK CHECKLIST

- ☐ 🔍 Find the wait cycle
- ☐ ⚠️ Handle SQLSTATE 40P01
- ☐ 🔄 Retry the whole transaction
- ☐ 📊 Bound attempts; backoff + jitter
- ☐ ⌚ Keep transactions short
- ☐ 👥 Lock shared rows consistently



ELLY'S FINAL TIP

A retry is part of the design—not permission to repeat unsafe side effects.



Better
Databases
Brighter
Tomorrows

OBSERVE
IMPROVE
KEEP GOING

NEGATIVE TEST:

Force the cycle and prove recovery does not duplicate work.

Resilient
Systems
Create
Happier
People



ELLYPUNK

ellypunk.com/training/sql-deadlocks-find-the-cycle-retry-the-transaction

© 2026 Ellypunk / NeetOps



1 FIRST PAGE

We fetch the first page in ID order using a stable key.

ID
10
20

Last key = **20**

2 A NEW ROW ARRIVES

A new row is inserted with ID 15. Using OFFSET 2 now repeats a row.

ID
10
15
20
30

OFFSET 2 repeats **20**.

3 CURSOR CONTINUES

Earlier inserts do not shift the next page.

ID
ID > 20
30

New rows after the cursor may appear.



REMEMBER THE LAST KEY.

Sort by a unique, unchanging key.

KEY IDEA:

First page returns IDs 10 and 20. Store **20** as the cursor. Next page asks for IDs after **20**.

SAME DATA. BETTER SYSTEMS.



CLEAN LOGIC
SCALABLE SYSTEMS
HAPPIER DEVELOPERS

1 PAGE 1 RETURNS 10 AND 20

We sort by a unique, unchanging key (ID in ascending order).

IDs (ascending)
ID
10
20

Last key = **20**

2 STORE 20 AS THE CURSOR

Store the last key from page 1. We saved **20** as the cursor.

Cursor
20

3 NEXT PAGE ASKS FOR IDs AFTER 20

The next page requests IDs with a key greater than the cursor (20).

Get next page:

IDs > **20**

4 RESULT STARTS AT 30

The result starts at 30, even if 15 was inserted. New rows before 20 do not affect the next page.

Next page results
ID
30
40



BUILD A RELIABLE CURSOR.

Stability comes from clear rules, not luck.

KEY IDEA:

A reliable cursor needs clear, consistent rules so results stay stable even while data changes.

BETTER QUERIES
BRIGHTER
TOMORROW.

Good Ideas
Better Data

CLEAN LOGIC
SCALABLE SYSTEMS
HAPPIER DEVELOPERS

1 ADD ID WHEN TIMESTAMPS TIE



If multiple items have the same timestamp, add ID to make the cursor unique and stable.

Created At	ID
2026-09-22 10:00	101
2026-09-22 10:00	102
2026-09-22 10:01	103

Order by Created At, then ID so the order is always unique.

2 KEEP FILTERS AND SORT ORDER CONSISTENT



Use the same filters and sort order for every page request.

Always keep:

- ✓ Same filter conditions
- ✓ Same sort order (e.g. Created At, ID)
- ✓ Same sort direction (e.g. ascending)

Changing filters or sort order can lead to missing or duplicate results.

3 INDEX THE ORDERED KEY



Index the ordered key (e.g. Created At, ID) so the database can find the next page efficiently.



An index on the ordered key helps keep pagination fast and scalable.

4 A LIVE CURSOR IS NOT A FIXED SNAPSHOT



A cursor reflects the current state of the data. New rows may appear after your last page.

A live cursor is NOT a fixed snapshot.

5 TREAT CURSOR TOKENS AS UNTRUSTED INPUT



Cursor tokens come from clients and can be tampered with. Treat them as untrusted input.

- ✓ Validate format
- ✓ Check expected fields
- ✓ Reject invalid tokens

Do not assume they are correct or safe.

6 LIVE RESULTS CAN CHANGE



A cursor is a position, not a fixed snapshot. Choose a snapshot when the user needs one unchanging result set.



Use a snapshot when you need a consistent, unchanging result set.

Better Queries
Brighter Tomorrow

WORKED EXAMPLE: INSERT BETWEEN PAGES.

A UNIQUE ORDER KEY KEEPS RESULTS STABLE.

PROBLEM:
Page 1 returns IDs 10 and 20.
Then ID 15 arrives.
OFFSET 2 would repeat 20.

Here's a worked example showing how cursor pagination stays stable even when new rows are inserted.

STABLE RESULTS
HAPPIER DEVELOPERS

CLEAN LOGIC
SCALABLE SYSTEMS
HAPPIER DEVELOPERS

Good Ideas
Better Data

1 INITIAL PAGE 1 RESULTS

We fetch the first page in ID order using a stable key. Page 1 returns IDs 10 and 20.

ID
10
20

Last key = 20

2 A NEW ROW ARRIVES

Then a new row is inserted with ID 15.

ID (sorted order)
10
15
20
30

The data now has IDs 10, 15, 20, 30.

3 OFFSET WOULD REPEAT 20

If we used OFFSET 2 now, it would return ID 20 again.

OFFSET 2 result
20
30

OFFSET 2 would repeat 20. This can lead to duplicates and unstable results.

4 CONTINUE AFTER KEY 20

Instead, we continue after the last key 20 using a unique, immutable order key. Page 2 starts at ID 30 — no repeat from this earlier insertion.

ID (after key 20)
30
...

ACTION:
Continue after key 20.

CHECK:
Page 2 starts at ID 30; no repeat from this earlier insertion.

IMPROVE:
Use a unique, immutable order key.

Later inserts may still appear.

RECAP: CONTINUE FROM A POSITION.

You now know
how cursor pagination
keeps results stable
while data changes.
Here's a quick recap
to remember the
key points!

BETTER
QUERIES
BRIGHTER
TOMORROW.
♡

KEY TAKEAWAYS

Small habits. Stable results.

- 1** Choose a unique, unchanging order.

- 2** Remember the last returned key.

- 3** Seek after that key on the next page.

- 4** Keep filters stable.

- 5** Decide whether a fixed snapshot is needed.

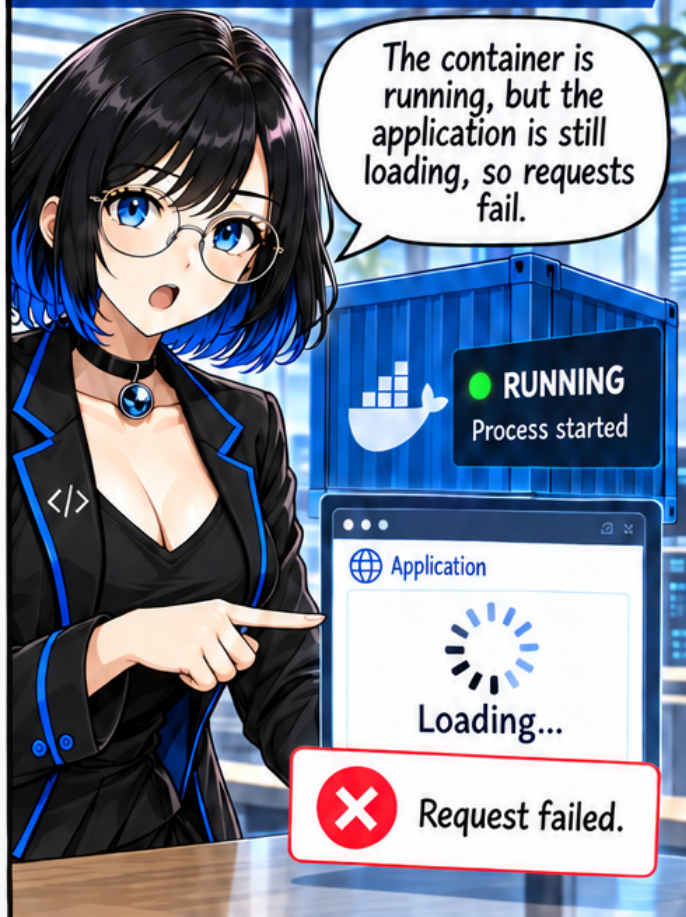

**Final tip
from Elly!**

Same
Ideas.
Deeper
Understanding.
♡

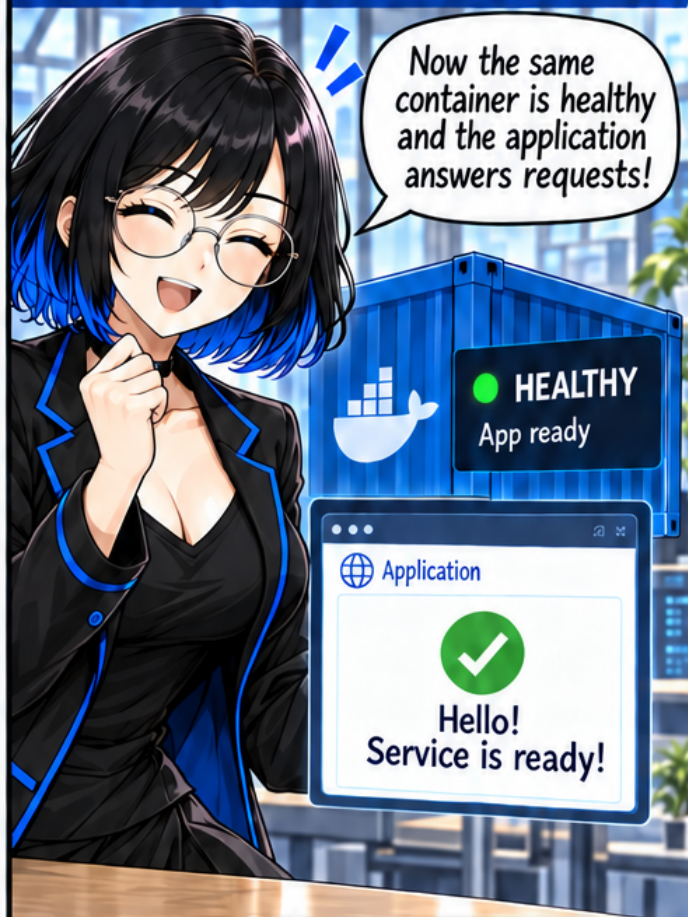
**A CURSOR IS A
POSITION, NOT A
PAGE NUMBER.**



RUNNING IS NOT READY.



PROBE THE SERVICE BEHAVIOR.



A301 • ADVANCED

 **NEETOPS**

23 September 2026

PAGE 2 / 5


BUILD
LEARN
SHIP
TOGETHER




THREE HEALTH STATES

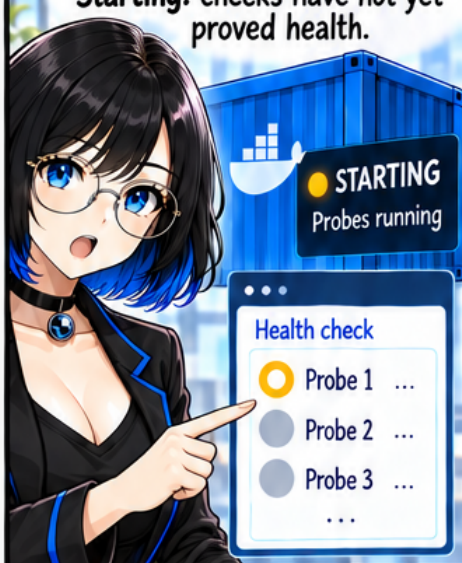
A container can be in one of three health states as probes run over time.

OBJECTIVE:
Understand what the three Docker health states mean.


HEALTHY
APPS
HAPPIER
USERS


1 **STARTING**

Starting: checks have not yet proved health.

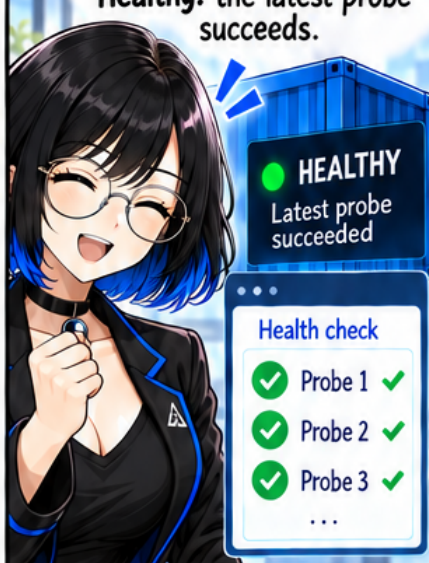



STARTING
Probes running

Health check
● Probe 1 ...
● Probe 2 ...
● Probe 3 ...
...

2 **HEALTHY**

Healthy: the latest probe succeeds.

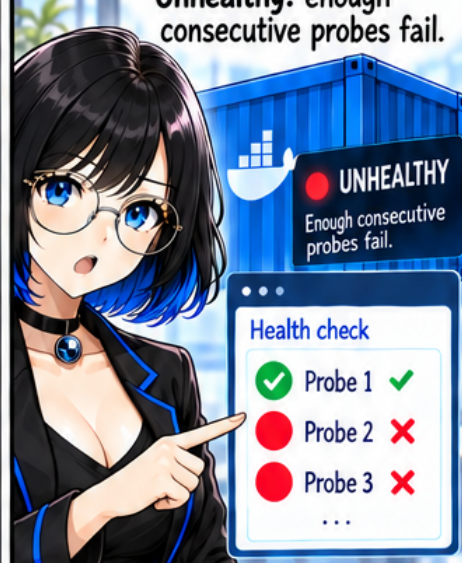



HEALTHY
Latest probe succeeded

Health check
✓ Probe 1 ✓
✓ Probe 2 ✓
✓ Probe 3 ✓
...

3 **UNHEALTHY**

Unhealthy: enough consecutive probes fail.




UNHEALTHY
Enough consecutive probes fail.

Health check
✓ Probe 1 ✓
● Probe 2 ✗
● Probe 3 ✗
...



Running and healthy answer different questions.


RUNNING
Process started
UNHEALTHY
Enough consecutive probes fail.

SAME CONTAINER. DIFFERENT ANSWERS.




OBSERVE
UNDERSTAND
SOLVE
BUILD
TOGETHER
♥

DESIGN THE PROBE

A simple check. A more reliable service.

OBJECTIVE:

Design an effective health probe for your application.

REAL
SYSTEMS
HEALTHIER
TOMORROWS
♥

1. TEST USEFUL APPLICATION BEHAVIOR, NOT ONLY A LIVE PROCESS.

A process can be running, but the service might still be broken.

- Service running
- Database connected
- Responding to requests

Service Response



Hello!
Service is healthy!

2. THE PROBE MUST USE TOOLS PRESENT INSIDE THE IMAGE.

Probe tool
(inside container)

- ✓ Check application endpoint
- ✓ Verify critical dependency
- ✓ Return success or failure

Your
Application

Use tools that are already in the image. Keep the probe simple, focused and reliable.

3. GIVE SLOW STARTUP A GRACE PERIOD; COUNT LATER FAILURES.



A health check reports state; routing and restart policies belong to your platform.

CONTAINERS
OBSERVABILITY
RELIABLE APPS

A301 • ADVANCED

 **NEETOPS** | 23 September 2026

PAGE 4 / 5

A good probe tests the service your users need.

WORKED EXAMPLE: SLOW START

OBJECTIVE:
Allow startup grace; health checks still run, but early failures do not count.

STABLE STARTUPS
HAPPIER USERS

BUILD
LEARN
SHIP
TOGETHER

CONTAINERS
OBSERVABILITY
RELIABLE APPS

Better Systems
Brighter Tomorrows.

CONTAINERS
SCALABLE SYSTEMS
REAL PROGRESS

1 PROBLEM:

The process starts before its service is ready.

The container is running, but the application is still starting, so requests fail.

RUNNING
Process started

Application
Loading...

Service not ready yet.

2 ACTION:

Probe an application response; allow startup grace.

Check the service's response and allow extra time for startup.

RUNNING
Process started

Application
Loading...

Within the startup grace period.

3 CHECK:

Status turns healthy once the probe succeeds.

The application is now responding and the probe succeeds!

HEALTHY
App ready

Application
Hello!
Service is ready!

Status turns healthy after a successful probe.

4 IMPROVE:

Alert on later unhealthy results and inspect probe logs.

If the service later becomes unhealthy, get an alert and inspect the probe logs.

UNHEALTHY
Probe failing

Application
Service unhealthy

Probe Logs

- Probe attempt failed
- Service not responding
- Timeout
- Investigate further



Let's recap what we've learned!

BETTER SYSTEMS
BRIGHTER TOMORROWS.
♡

HEALTH CHECK RECAP

- 1 Test application behavior.
- 2 Give startup grace without hiding later failures.
- 3 Observe health status and probe output.
- 4 Include the probe tool in the image.

HEALTHY APPS
HAPPIER USERS
♡

CONTAINERS
OBSERVABILITY
RELIABLE APPS



CONTAINERS
SCALABLE SYSTEMS
REAL PROGRESS

FROM PROCESS TO PROBE TO RESULT

RUNNING PROCESS



The container is running, and the application is initializing.

HEALTH PROBE



The platform periodically probes the application.

OBSERVED RESULT



Use the result to decide what happens next.





SQL ISOLATION LEVELS:

WHAT EACH TRANSACTION CAN SEE

OBJECTIVE:

Compare visibility and concurrency in SQL Server.

Understand
Solve
Build
Better



1. TWO TRANSACTIONS

Transaction A

Both sessions read the same row.

READ

Shared Row		
ID	Name	Balance
1	Alice	100

READ

Transaction B

Both sessions read the same row.

2. ONE SESSION CHANGES DATA

One session changes data.

ID	Name	Balance
1	Alice	200

3. WHAT CAN THE OTHER SEE?

What may the other session read?



Isolation changes the answer.

Different isolation levels allow different visibility, concurrency and trade-offs.

Reliable
Data
Stronger
Systems
Together



Consistent Data.
Clear Stories.
A Brighter Tomorrow.

READ COMMITTED vs REPEATABLE READ

SAME ROW. DIFFERENT RESULTS.

Isolation levels control what your statements can see.

Understand Solve Build Better

1. READ COMMITTED

Each statement sees the latest committed data.



2. REPEATABLE READ

Keeps shared locks on read rows to prevent other transactions from changing them.



A new matching row may still appear under **REPEATABLE READ**.

READ COMMITTED may use row versions if the database option is enabled.



Same data.
Different
isolation levels.
Different
results.



SNAPSHOT OR SERIALIZABLE?

UNDERSTAND WHAT
EACH TRANSACTION CAN SEE
IN SQL SERVER.

Isolation
Builds
Stronger
Systems



SNAPSHOT

SNAPSHOT reads the transaction-start version.
It must be enabled; write conflicts may occur.

Transaction A
(at start)

My snapshot contains:

START

ID	Name	Balance
	Alice	
	Bob	

Transaction B
(inserts a new row)

Insert Carol and commit.

COMMIT

ID	Name	Balance
	Alice	
	Bob	
	Carol	

Transaction A
(still the same view)

I still see only my snapshot:

ID	Name	Balance
	Alice	
	Bob	

SERIALIZABLE

SERIALIZABLE prevents matching inserts with key-range locks.
More blocking may follow.

Transaction A
(selects a range)

I read a key range
(e.g. names from
A to M).

Key-range lock
protects the range

Names
A to M

Transaction B
(attempts a matching insert)

This insert matches
the protected range,
so it must wait.

Insert
Carol
(WAITING)

After Transaction A commits

Once I commit,
Transaction B
can proceed.

Range lock
released

Insert
Carol
(now allowed)

Choose the
anomaly you
must prevent.

Isolation levels are trade-offs
between concurrency,
consistency and business needs.

Clearer
Data.
Brighter
Tomorrow.



A301 • ADVANCED

24 September 2026

PAGE 4

WORKED EXAMPLE: STOP A PHANTOM

ISOLATION LEVELS

CONCURRENCY

RELIABLE SYSTEMS

A BRIGHTER TOMORROW





Isolation Turns Theory Into Confidence.



1. PROBLEM

Under REPEATABLE READ, a new matching row can appear on a later search.

Transaction A
(REPEATABLE READ)

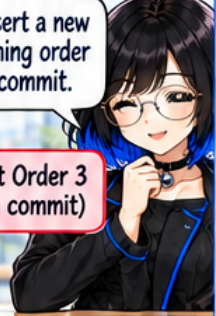


First search finds 2 orders for this customer.

Orders
(customer_id = 42)

ID	Customer	Amount
1	42	100
2	42	50

Transaction B



I'll insert a new matching order and commit.

Insert Order 3 (then commit)

Transaction A
(later search)



Now I see 3 orders!

Orders
(customer_id = 42)

ID	Customer	Amount
1	42	100
2	42	50
3	42	75

2. ACTION

For this count-and-insert rule, use **SERIALIZABLE**.

Transaction A
(SERIALIZABLE)



New attempt. I'll read the matching orders using **SERIALIZABLE**.

Orders
(customer_id = 42)

ID	Customer	Amount
1	42	100
2	42	50

I still see 2 orders.

Transaction B



I'll insert a matching order...

Order 3 (pending)

 **WAITING**
(for A to commit)

Transaction A
(still open)



While A is open, I still see 2 orders.

Orders
(customer_id = 42)

ID	Customer	Amount
1	42	100
2	42	50

4. IMPROVE

Keep the transaction short; review the search index.

Transaction A



I'm done. Committing now.

Transaction B
(can proceed)



Great! Now my insert can complete.

Orders
(customer_id = 42)

ID	Customer	Amount
1	42	100
2	42	50
3	42	75

Correct isolation prevents phantoms.

Short transactions.
Good indexes.
Stronger systems.
Together. 



RECAP:

CHOOSE THE GUARANTEE

Understand
Solve
Build
Better

READ COMMITTED

committed data;
later reads may
change.

REPEATABLE READ

protects rows
read; phantoms
remain possible.

SNAPSHOT

transaction-start
view; enable it
first.

SERIALIZABLE

protects searched
ranges; expect
more blocking.

Start with the anomaly
your rule cannot tolerate.

Reliable
Data
Stronger
Systems
Together

SQL INDEXES

Choose a Key for a Real Query

OBJECTIVE:

Match one busy query to a narrow index, then check the plan and write cost.

Faster Queries
Happier People
♥

Clear Data
Brighter Tomorrow
♥

Data
Indexes
Queries
Better Systems

1. A BUSY QUERY

This query finds orders for a specific customer in a recent date range.

Orders		
OrderId	CustomerId	OrderDate

2. AN INDEX HELPS

A narrow index on CustomerId and OrderDate helps the database quickly locate the matching rows.

Index	
CustomerId	OrderDate

Only the matching rows are read.

3. CHECK THE PLAN AND WRITE COST

After adding the index, check the query plan to confirm it's used and write down the cost.



Right index.
Real results.
Brighter tomorrow.
♥

A301 • ADVANCED

25 September 2026

PAGE 2

Real Queries
Better Indexes
Happier People
♥

Start with the real query, then choose the key order.

Right Index.
Real Progress.
♥

✓ Filter first.
✓ Narrow fast.
✓ Find the rows.
✓ Brighter tomorrow. ♥

Indexes
Query Plans
Data Design
Better Systems

Clear Data
Brighter Tomorrow
♥

1. EQUALITY FILTER: CustomerId

First, apply the equality filter on CustomerId to narrow the data to one customer.

Filter: CustomerId = 1001

CustomerId	OrderDate	...
1001	2026-09-01	...
1001	2026-09-05	...
1001	2026-09-12	...
...	...	...

2. DATE RANGE: OrderDate

Then apply the date range on OrderDate to find the rows in the period you need.

Filter: OrderDate range

CustomerId	OrderDate	...
1001	2026-09-01	...
1001	2026-09-05	...
...	...	...

3. CANDIDATE KEY: (CustomerId, OrderDate)

A composite index on (CustomerId, OrderDate) keeps matching rows together and ordered by date.

Candidate index key

CustomerId → OrderDate

Equality filter first column (narrows to one customer)

Range filter second column (returns rows in order)

HOW THE INDEX KEEPS ROWS SORTED

CustomerId	OrderDate	...
1001	2026-09-01	...
1001	2026-09-05	...
1001	2026-09-12	...
...	...	...
1002	2026-09-02	...
1002	2026-09-08	...
1002	2026-09-20	...
...	...	...

Rows are grouped by CustomerId and ordered by OrderDate within each customer.

This index is a candidate to measure with your real query and plan.

Same idea. Deeper skills. A brighter tomorrow. ♥

READ GAIN, WRITE COST

Pick the right index. Faster reads, manageable writes.

Good
Indexes
Real
Impact
♡

NARROW KEY

Use only the columns the search needs.



**FASTER
READS**

Less data
to search.
Higher
performance.
♡

COVERING CHOICE

Included columns may
avoid lookups, but
increase index size.



**FEWER
LOOKUPS
(FASTER READS)**

**LARGER
INDEX SIZE
(MORE STORAGE)**

Sometimes
bigger
is worth it
♡

WRITE COST

Each extra index
adds work to inserts
and updates.



MORE INDEXES



**MORE WORK
ON WRITES
(SLOWER INSERTS)
AND UPDATES)**

Read fast.
Write smart.
Find the
balance.
♡

Check existing indexes
before adding one.



**READ
SPEED**



**STORAGE
AND WRITES**

Better
Systems.
Brighter
Tomorrow.
♡

A301 • ADVANCED

25 September 2026

PAGE 4

Same Data. Clearer Answers. ♥

Indexes Queries Better People ♥

TEST ONE CANDIDATE

Try an Index, Measure, Decide

Test Carefully. Better Systems ♥

Indexes Queries Better People ♥

Ideas Indexes Impact ♥

Data Indexes Queries Better Systems

1. PROBLEM

Orders lookup by CustomerId and recent OrderDate is slow.

Orders (sample rows)

OrderId	CustomerId	OrderDate
50123	1001	2026-09-20
50124	1001	2026-09-21
50125	1001	2026-09-22
50126	1002	2026-09-22
50127	1001	2026-09-23
50128	1003	2026-09-23
...	...	...

2. ACTION

Try key (CustomerId, OrderDate) in a safe test environment.

Candidate Index (sorted)

CustomerId	OrderDate
1001	2026-09-20
1001	2026-09-21
1001	2026-09-22
1001	2026-09-23
1002	2026-09-22
1003	2026-09-23
...	...

Sorted by key

3. CHECK

Compare the actual plan, logical reads, and write latency.

Actual Plan

Before (no index)

After (with index)

Logical Reads

Write Latency

Compare logical reads before and after.

Check the impact on write latency (insert/update).

4. IMPROVE

Keep it only if the workload benefits overall.

Measure. Understand. Make Better Choices. ♥

Faster Reads

Write Cost

Balance for your workload.

INDEX DECISION CHECKLIST

Better
Systems
Brighter
Tomorrow.
♡

A good index serves
a real workload.

- 1  Start with the query predicate.
- 2  Choose a narrow key order.
- 3  Compare actual plans and reads.
- 4  Include write and storage cost.

Keep the index
that earns
its upkeep.
♡

Understand
Compare
Choose
Build Better
♡

Good
Indexes
Real
Impact
♡