Diving Down the Concurrency Rabbit Hole

Mike Acton
macton@insomniacgames.com

Expectations...

- Start to unlearn "traditional" approach to concurrency issues
- Assume some knowledge of concurrency approaches

Expectations...

- Provide the rationale behind alternative approaches
- Understanding why data/solutions are fundamentally different.

Expectations...

- First in a series of concurrency optimization talks
- Later talks provide examples
- Later talks provide common patterns and solutions
- Later talks focus a lot on lock/wait-free techniques

Expectations...

- There are "thinking points" in here
- Unanswered questions as exercises

Expectations...

- I like to go off on interesting tangents.
- But you may already know that.

Let's Start w/
a Well-known
"problem" and
work backward...

Reference Problem

But replacing locks wholesale by writing your own lock-free code is not the answer. Lock-free code has two major drawbacks. First, it's not broadly useful for solving typical problems—lots of basic data structures, even doubly linked lists, still have no known lock-free implementations.

Lock-Free Code: A False Sense of Security
Herb Sutter
http://www.ddj.com/cpp/210600279

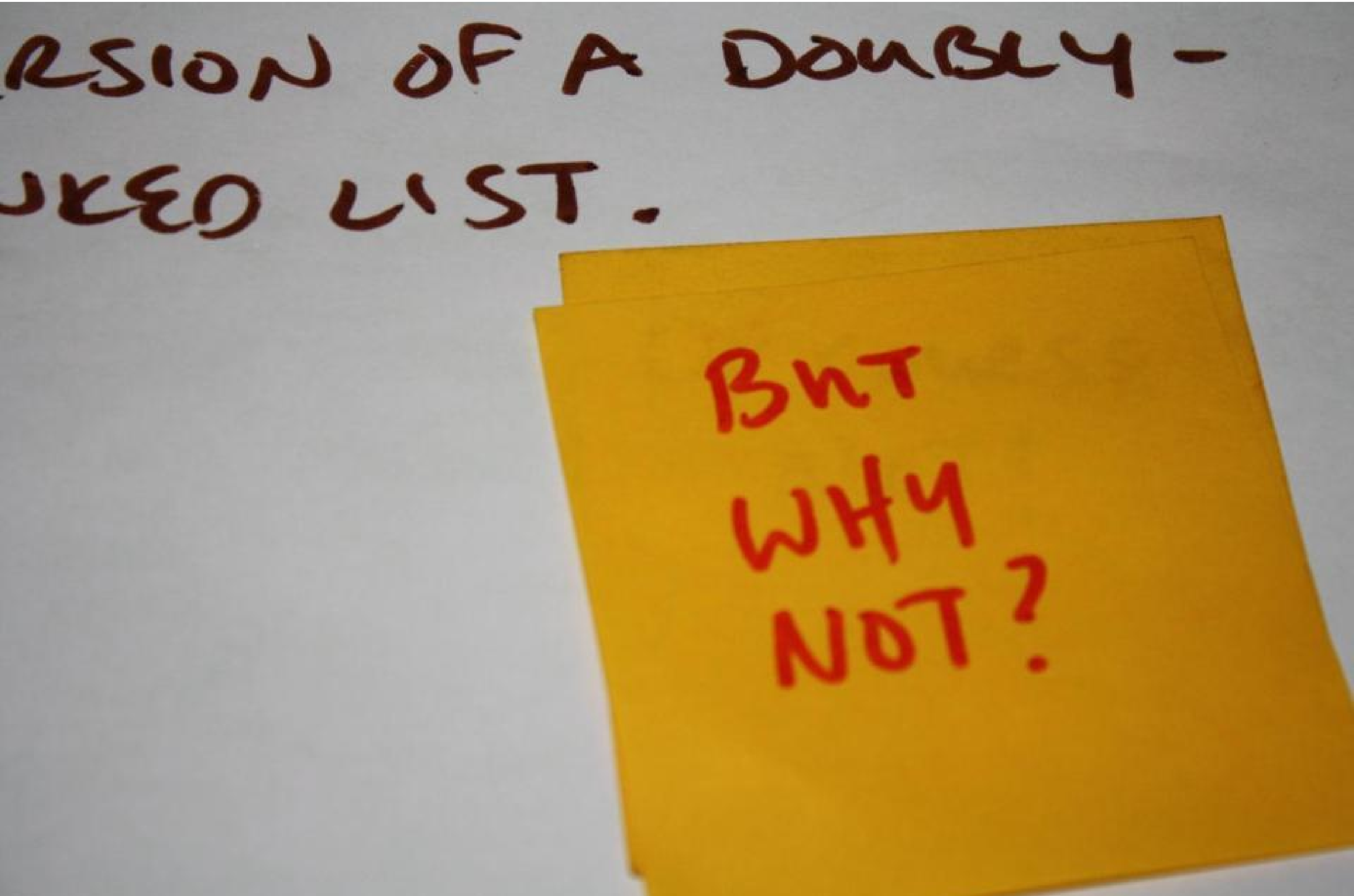
PROBLEM:

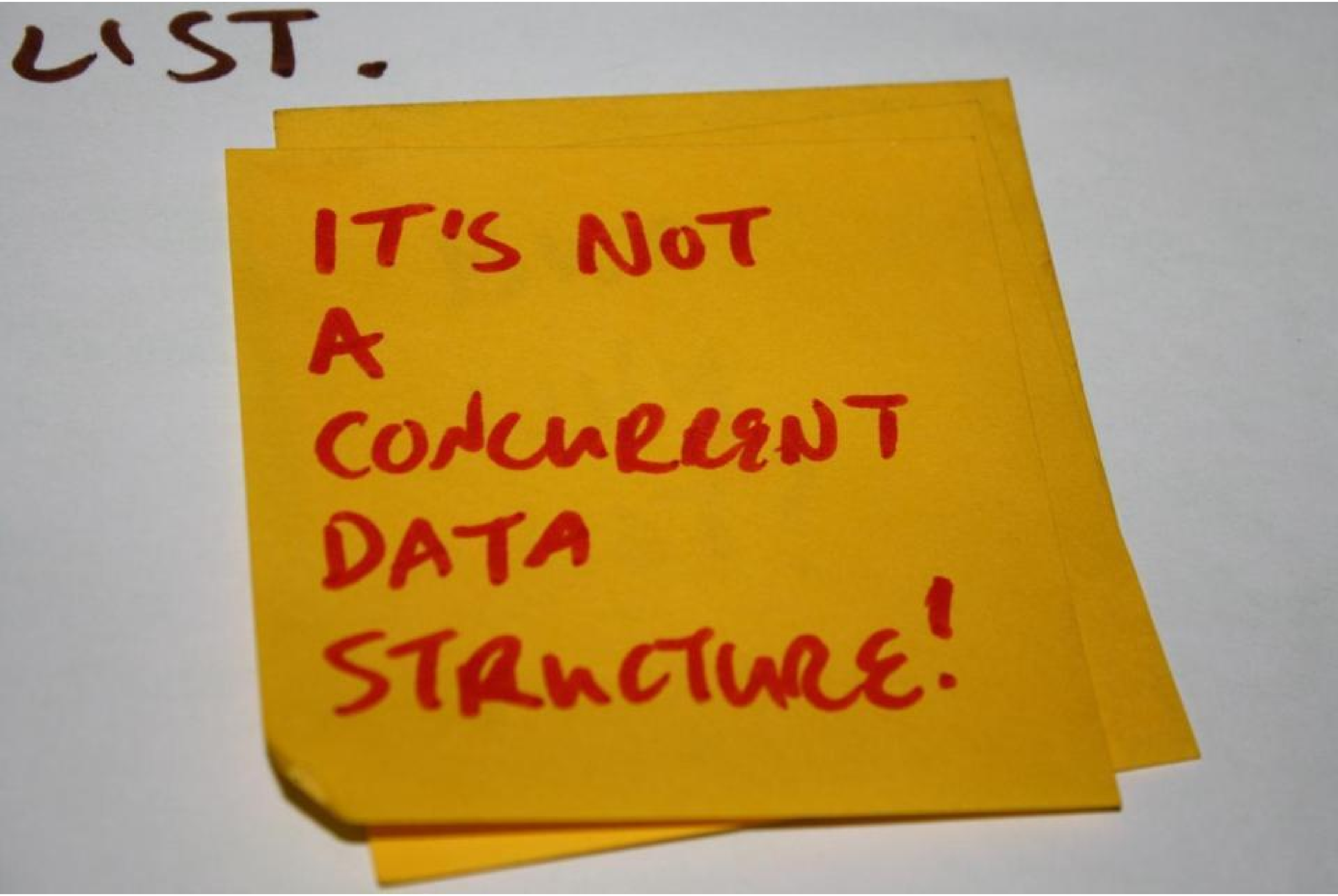
THERE IS NO LOCK-FREE VERSION OF A DOUBLY-LINKED LIST.

PROBLEM:

THERE IS NO LOCK-FREE VERSION OF A DOUBLY-LINKED LIST.

OF COURSE NOT!


RSION OF A DOUBLY-
NKED LIST.
BUT
WHY
NOT?


LIST.
IT'S NOT
A
CONCURRENT
DATA
STRUCTURE!

Reference Problem

But replacing locks wholesale by writing your own lock-free code is not the answer. Lock-free code has two major drawbacks. First, it's not broadly useful for solving typical problems—lots of basic data structures, even doubly linked lists, still have no known lock-free implementations.

Lock-Free Code: A False Sense of Security
Herb Sutter
http://www.ddj.com/cpp/210600279

As an aside, I disagree with the point above. But that's a topic for a different day.

Reference Problem

- The reference problem is just for context.
- No lock-free doubly-linked list here.
- Rather, background on why it's **not** an important problem.

Reference Problem

Should expect to understand:

- A doubly-linked list will not meet real constraints of a concurrent system.
- i.e. It's not going to be the solution/data to a concurrent problem.
- If it's used, it's only in a local context.

Reference Problem

Should expect to learn:

- Why and how concurrent data design is different.

PROBLEM:

THERE IS NO LOCK-FREE VERSION OF A DOUBLY-LINKED LIST.

IT'S NOT REALLY A PROBLEM.

PROBLEM:

THERE IS NO LOCK-FREE VERSION OF A DOUBLY-LINKED LIST.

MORE LIKE AN INTERESTING PUZZLE.

PROBLEM:

THERE IS NO LOCK-FREE VERSION OF A DOUBLY-LINKED LIST.

IT'S THE WRONG LEVEL OF ABSTRACTION FOR CONCURRENCY.

Why would data structures be different for concurrent designs?

CREATING CONCURRENT
DATA STRUCTURES
REQUIRES AN EXTRA
DIMENSION OF INFO

CREATING CONCURRENT
DATA STRUCTURES
REQUIRES AN EXTRA
DIMENSION OF INFO

THIS SEEMS
OBVIOUS

Why would data structures change?

- Doubly-linked lists solve a particular set of problems
- The concurrent "version" is a different problem
- Data is designed around the problem(s) being solved.

DATA STRUCTURES
REQUIRES AN EXTRA
DIMENSION OF INFO

THIS SE
O BUI

SOMETIMES.
TRAD. DATA
STRUCTS CAN
BE USED,

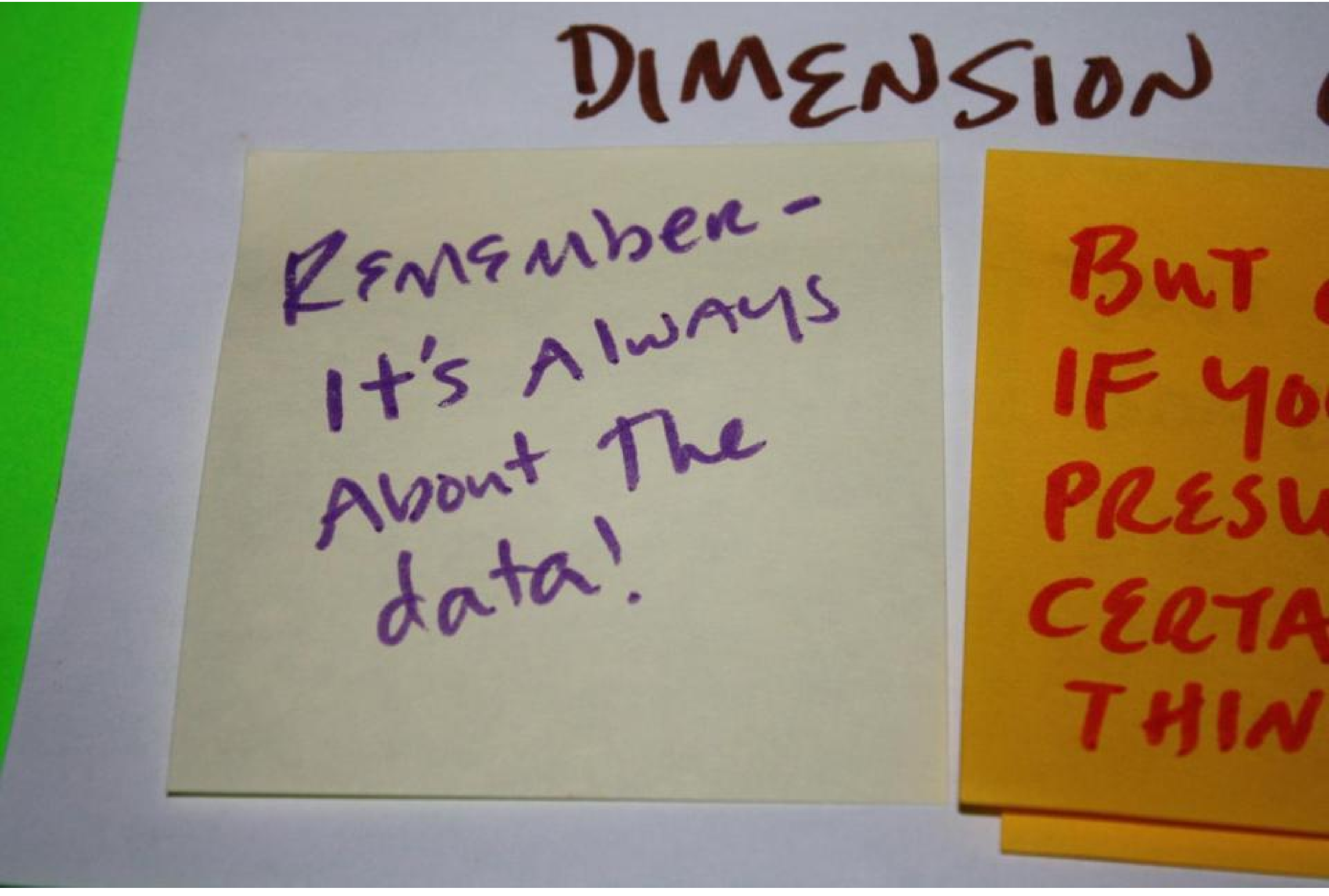
DATA STRUCTURES
REQUIRES AN EXTRA
DIMENSION OF INFO

THIS

JUST LIKE
SOMETIMES
2D STRUCTS
CAN BE USED
IN 3D APPS.

DIMENSION OF INFO

BUT ONLY
IF YOU
PRESUME
CERTAIN
THINGS.


DIMENSION
Remember -
It's Always
About The
data!
BUT

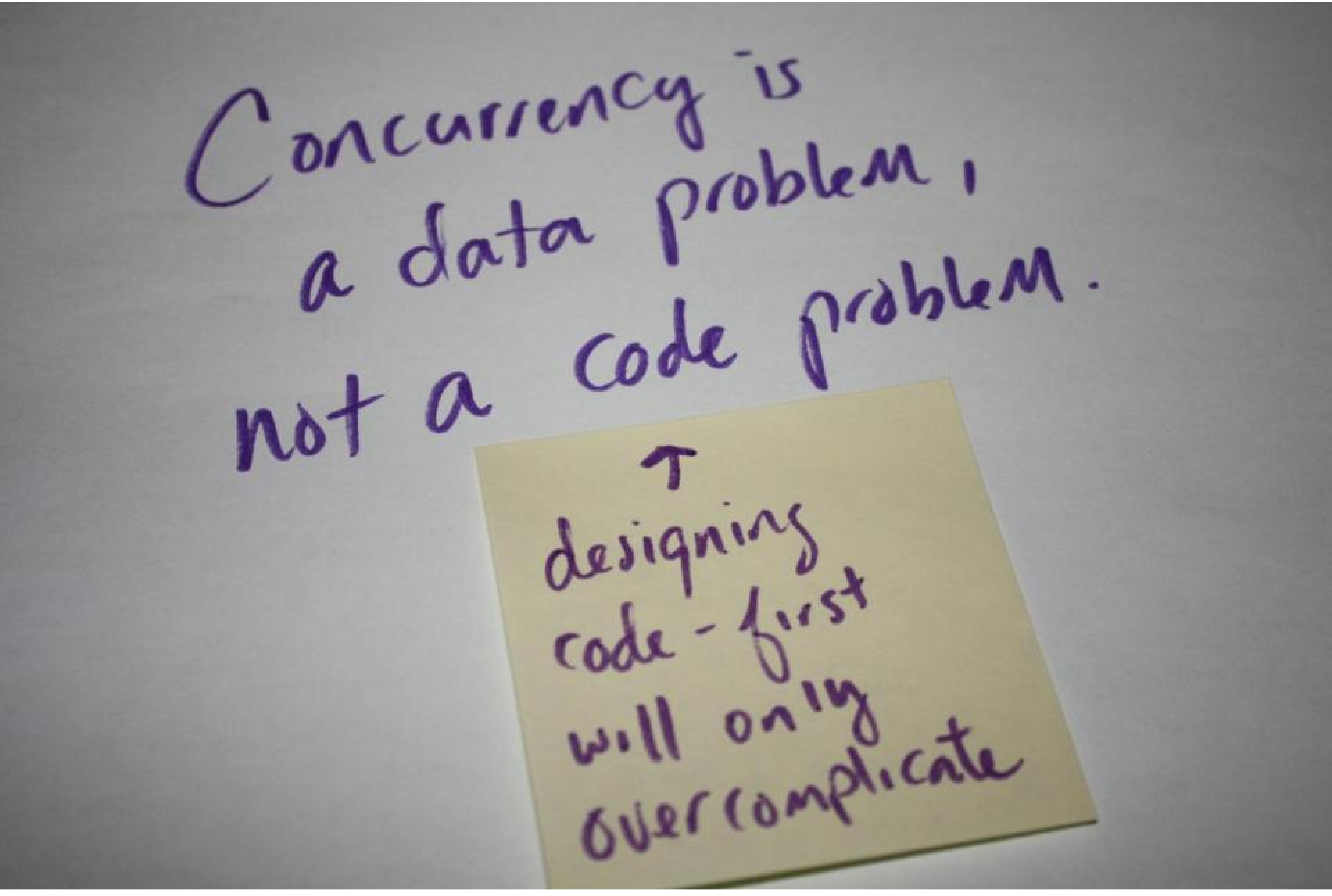
It's always about the data!

I will repeat this point a lot.

Why?

Because it's important!

Concurrency is
a data problem,
not a code problem.


Concurrency is
a data problem,
not a code problem.
designing
code-first
will only
overcomplicate

The Question

Is something like this the best data fit for *any* concurrency problem?

```
struct Node
{
    Node*   next;
    Node*   prev;
    Packet* data;
};
```

The data structure itself implies a different kind of problem (i.e. local)

DOUBLY-LINKED LIST PRESUMES SEQUENTIAL ORDER.

DOUBLY-LINKED LIST PRESUMES SEQUENTIAL ORDER.

Well, obviously it's a definition of an order, ...

Defining Order

```
struct Node
{
    Node*   next; <-- Defines an order
    Node*   prev; <-- (That's the point.)
    Packet* data;
};
```

ESUMES SEQUENTIAL
DER.

But xforms
of the data
are also
implicitly
ordered.

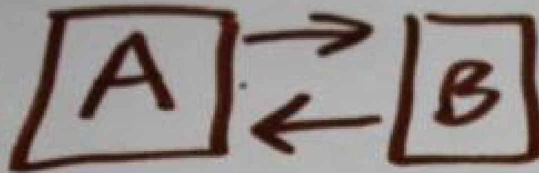
Transform Order

```
struct Node
{
    Node*   next; <-- But WHY is it defined
    Node*   prev; <-- this way in 1st place?
    Packet* data;
};
```

Transform Order

```
struct Node
{
    Node*   next; <-- But WHY is it defined
    Node*   prev; <-- this way in 1st place?
    Packet* data;
};
```

- It's to make certain operations easier.
- And give those operations certain properties.
- e.g. Insert, Delete



e.g.
INSERT

- INSERT (C) After (A)
- INSERT (D) After (A)

HAS GUARANTEED RESULT:

Transform Order

```
struct Node
{
    Node*   next; <-- But WHY is it defined
    Node*   prev; <-- this way in 1st place?
    Packet* data;
};
```

e.g. Insert

So that sequential insert instructions would:

- Have constant insert time
- Have a guaranteed (predictable) result
- Could be inserted before or after given any node
- etc.

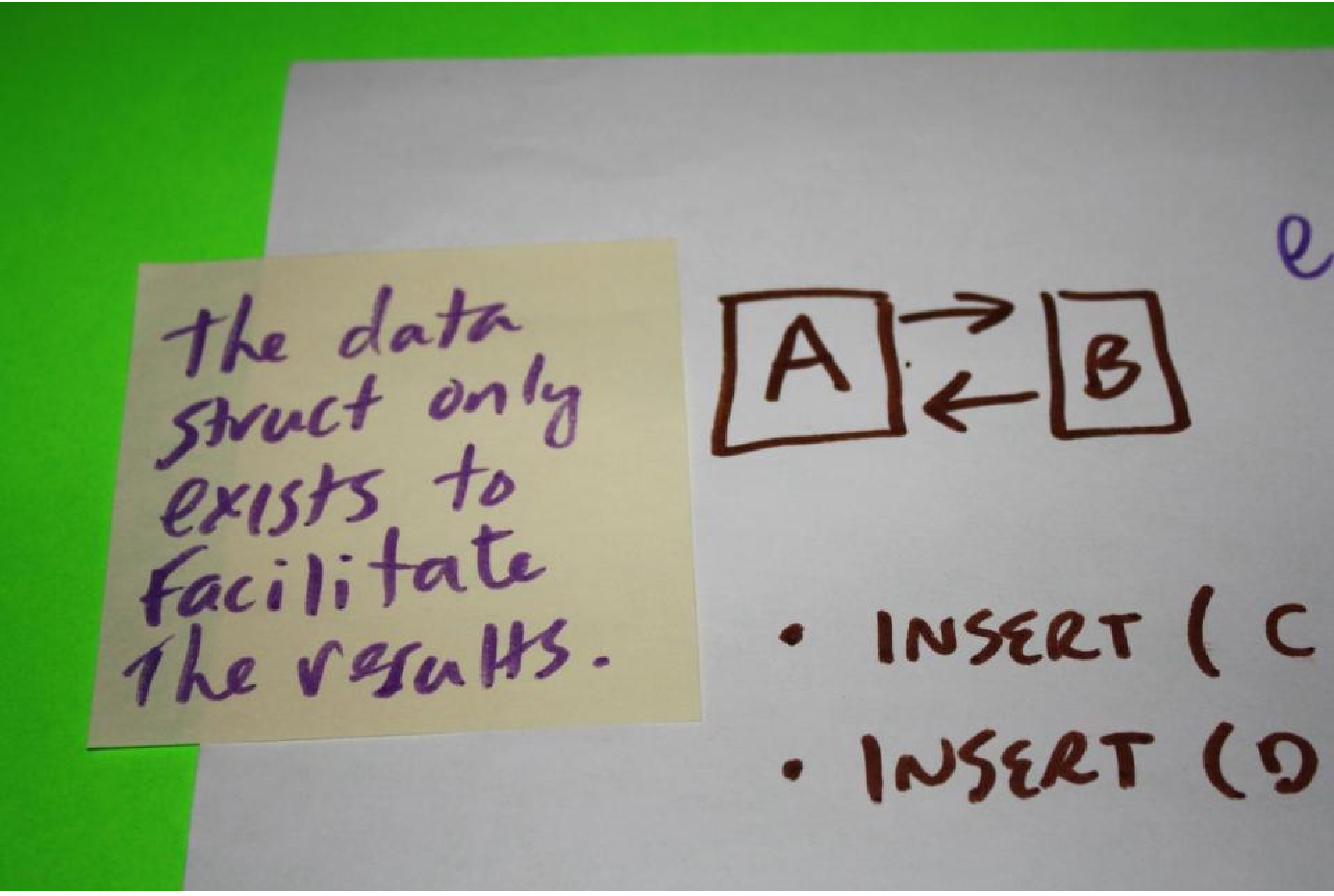
e.g
INSERT

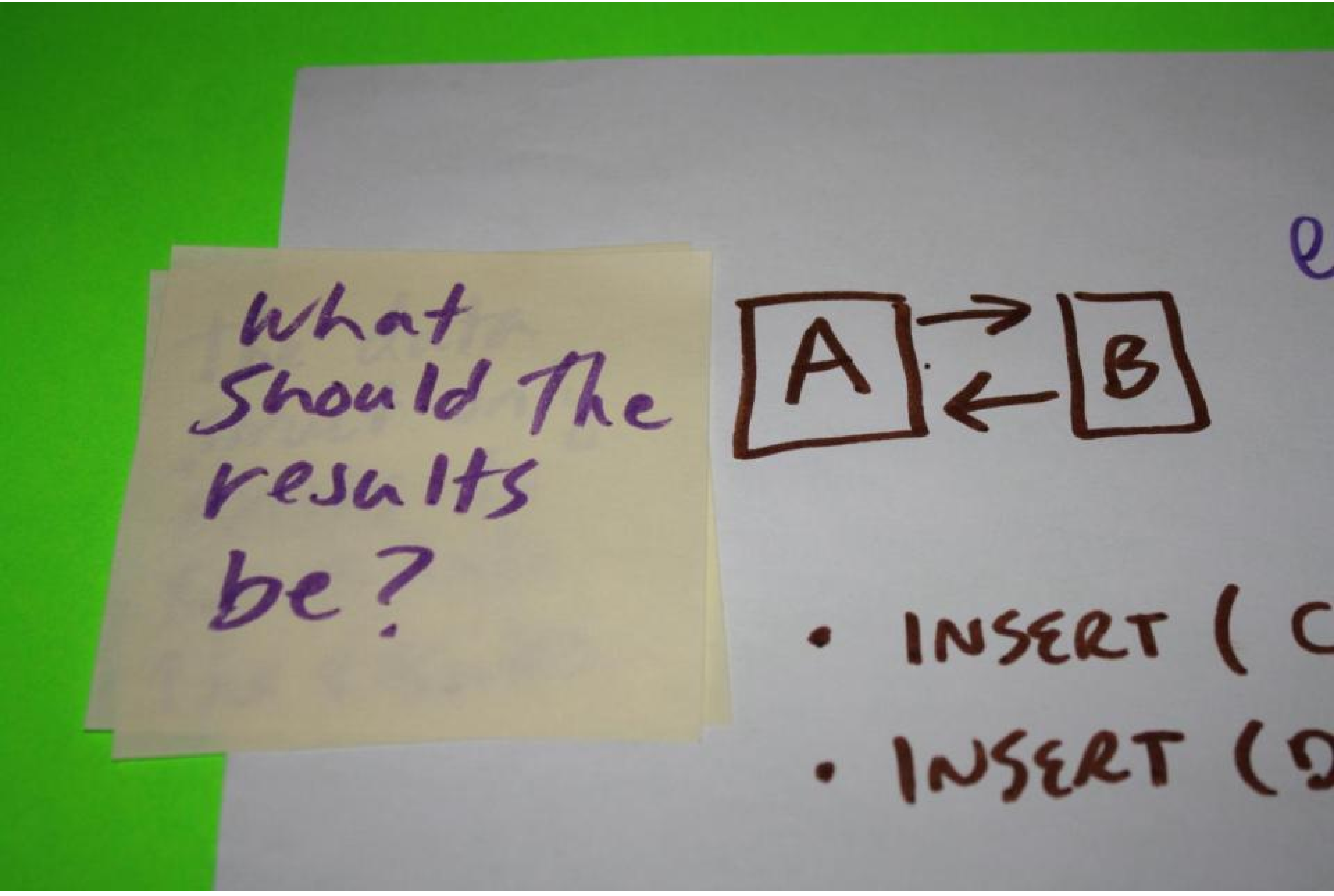
$\boxed{B}$

All ops
have
predictable
results

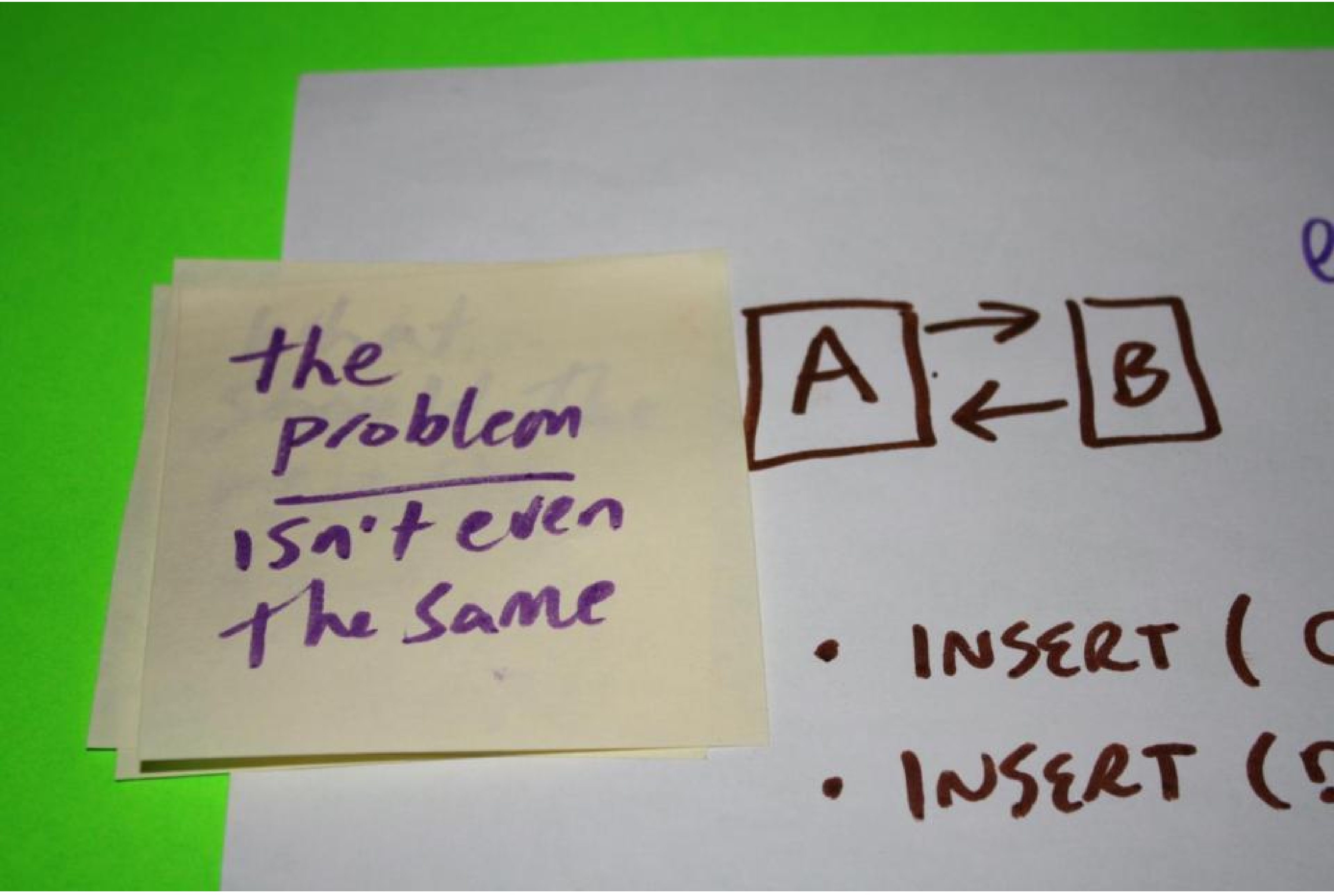
RT (C)

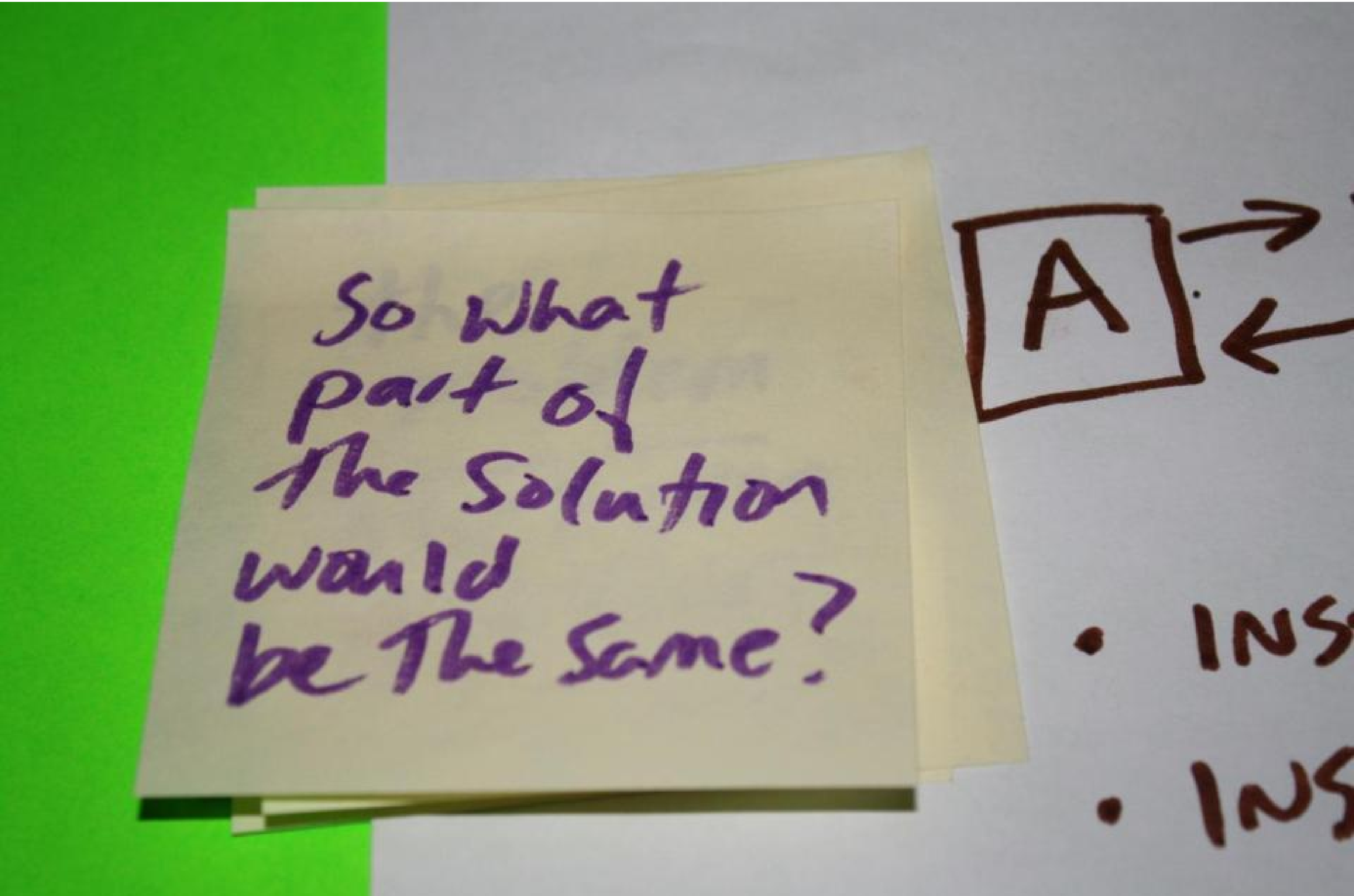
T (D)


the data struct only exists to facilitate the results.
A
B
e
• INSERT (C
• INSERT (D


what should the results be?
A
B
• INSERT (C
• INSERT (D

Need to define what the transformations must do before you can define what the data is.


the problem
isn't even
the same
A
B
• INSERT (
• INSERT (

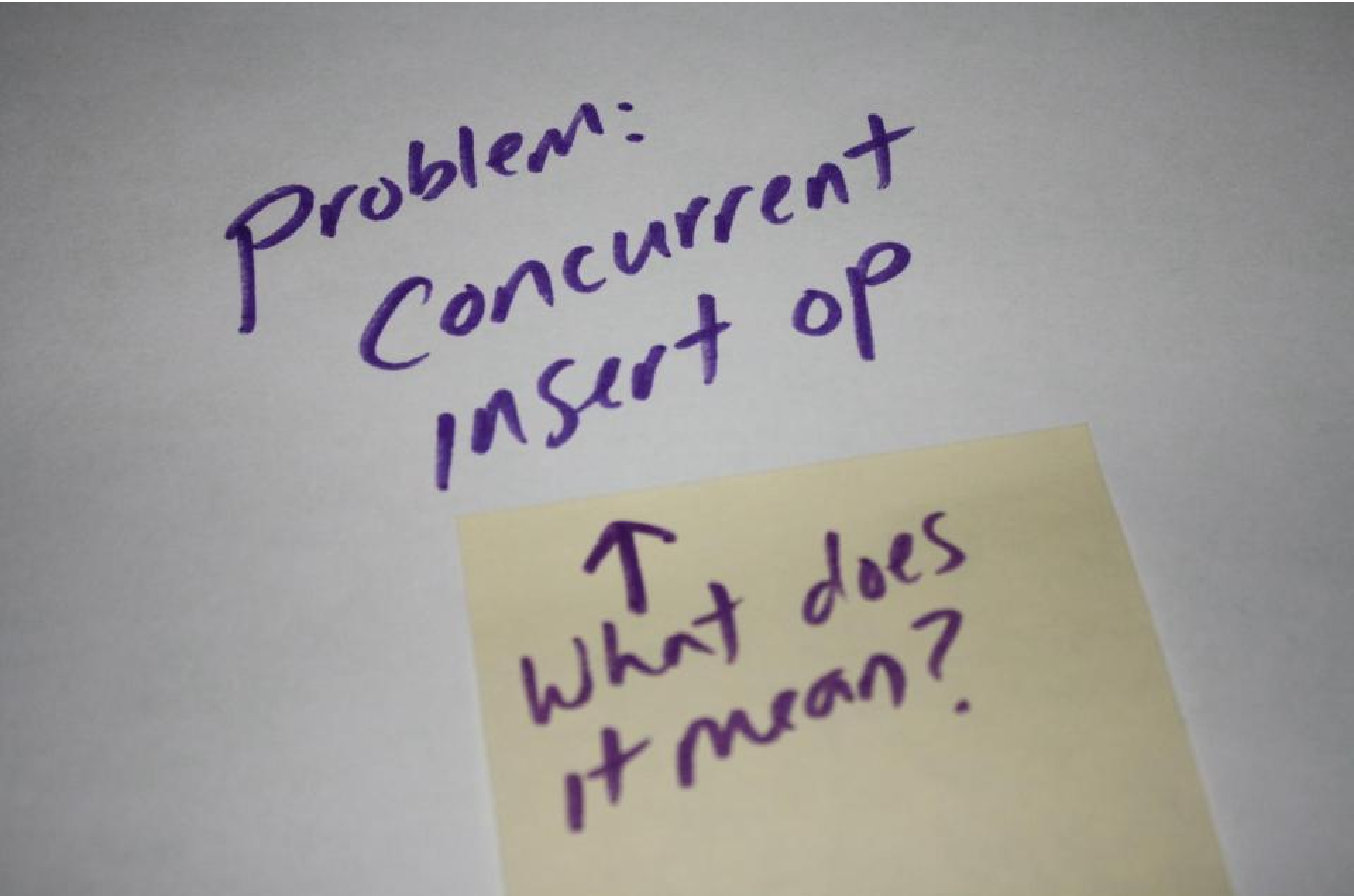

So What part of the Solution would be The Same?
A
• INS
• INS

Operations in a concurrent system would not have the same meaning.

The properties of those operations (constraints) would also be different.

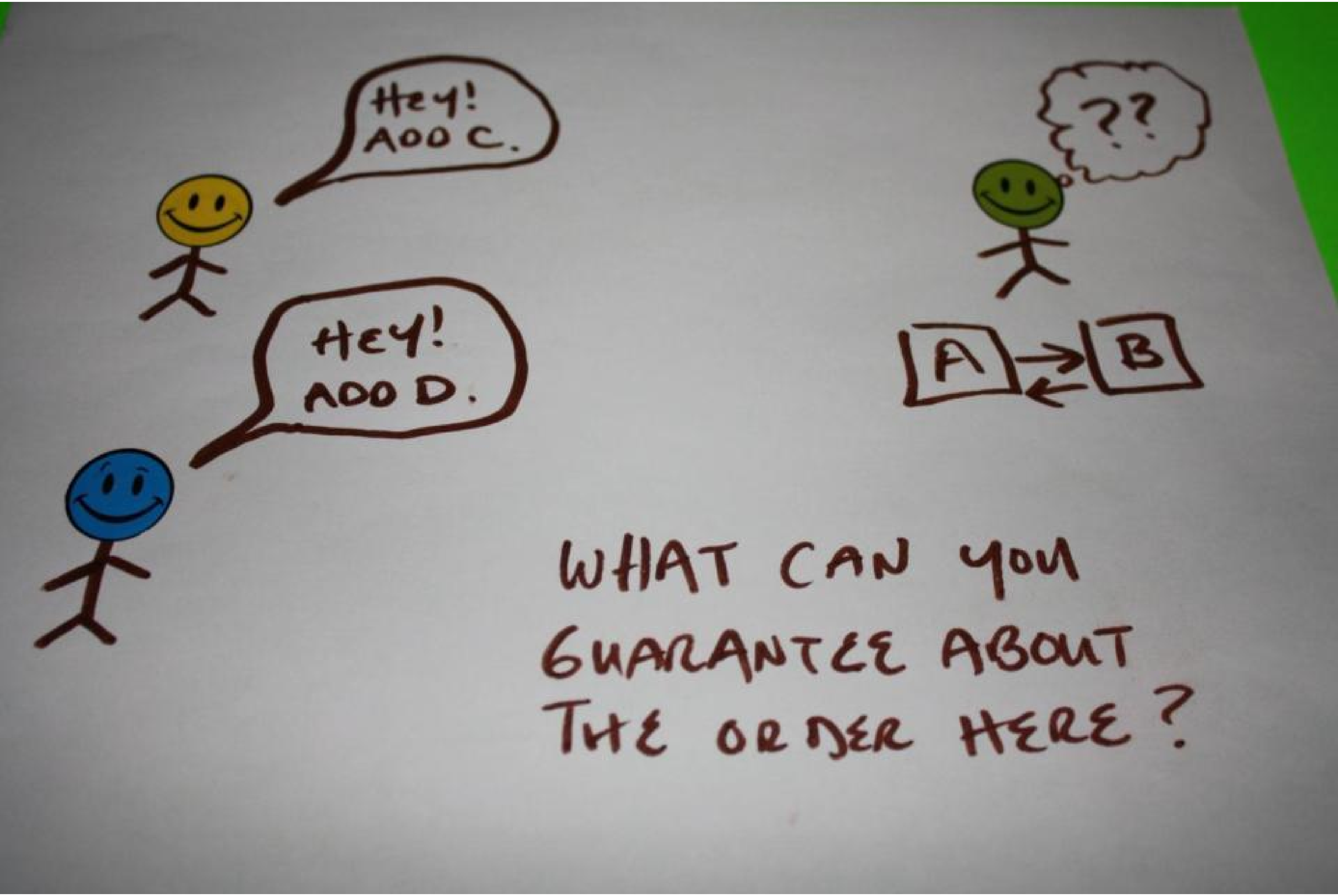
Let's look at the "same" problem as a conccurent operation...

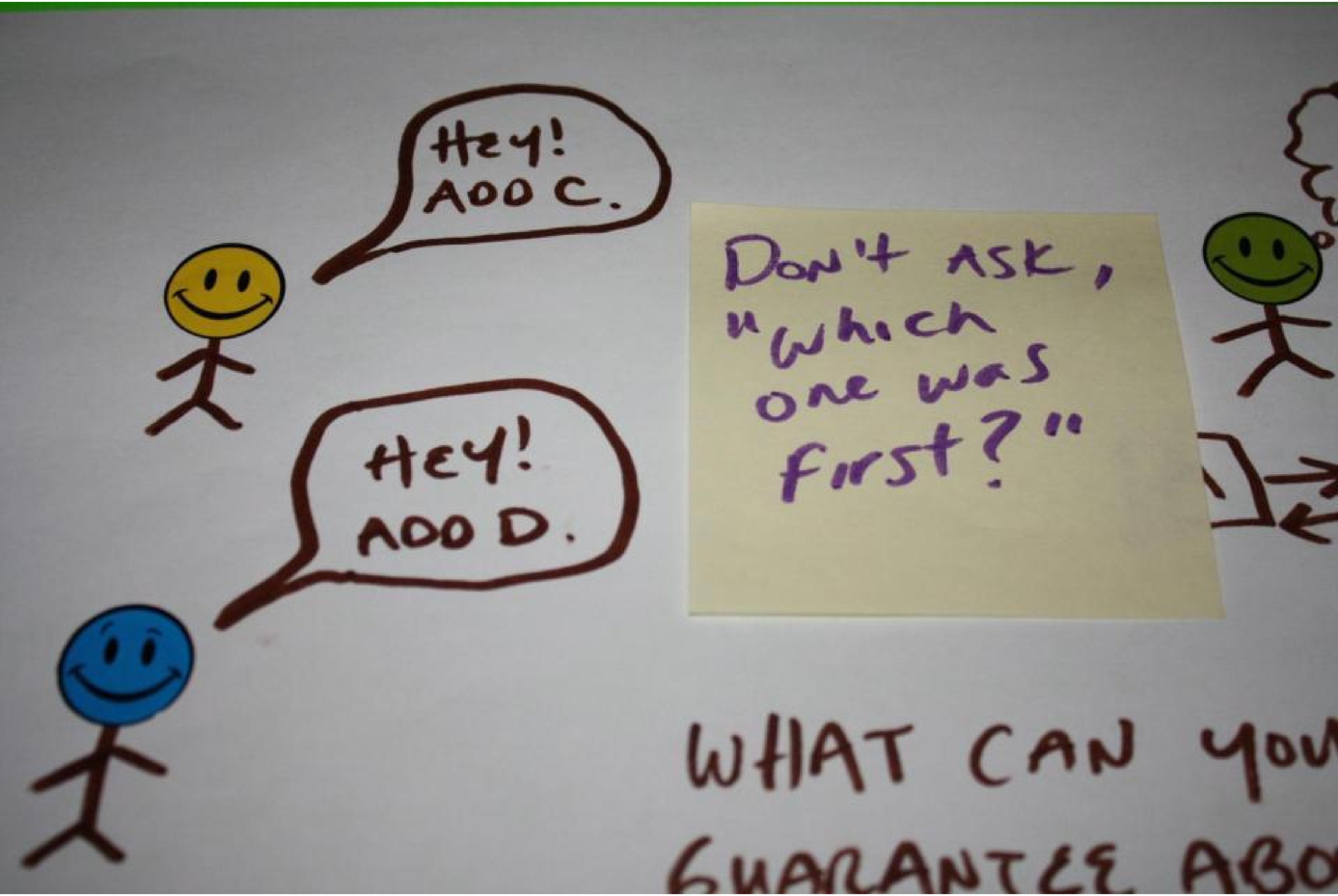
Problem:
Concurrent
insert op

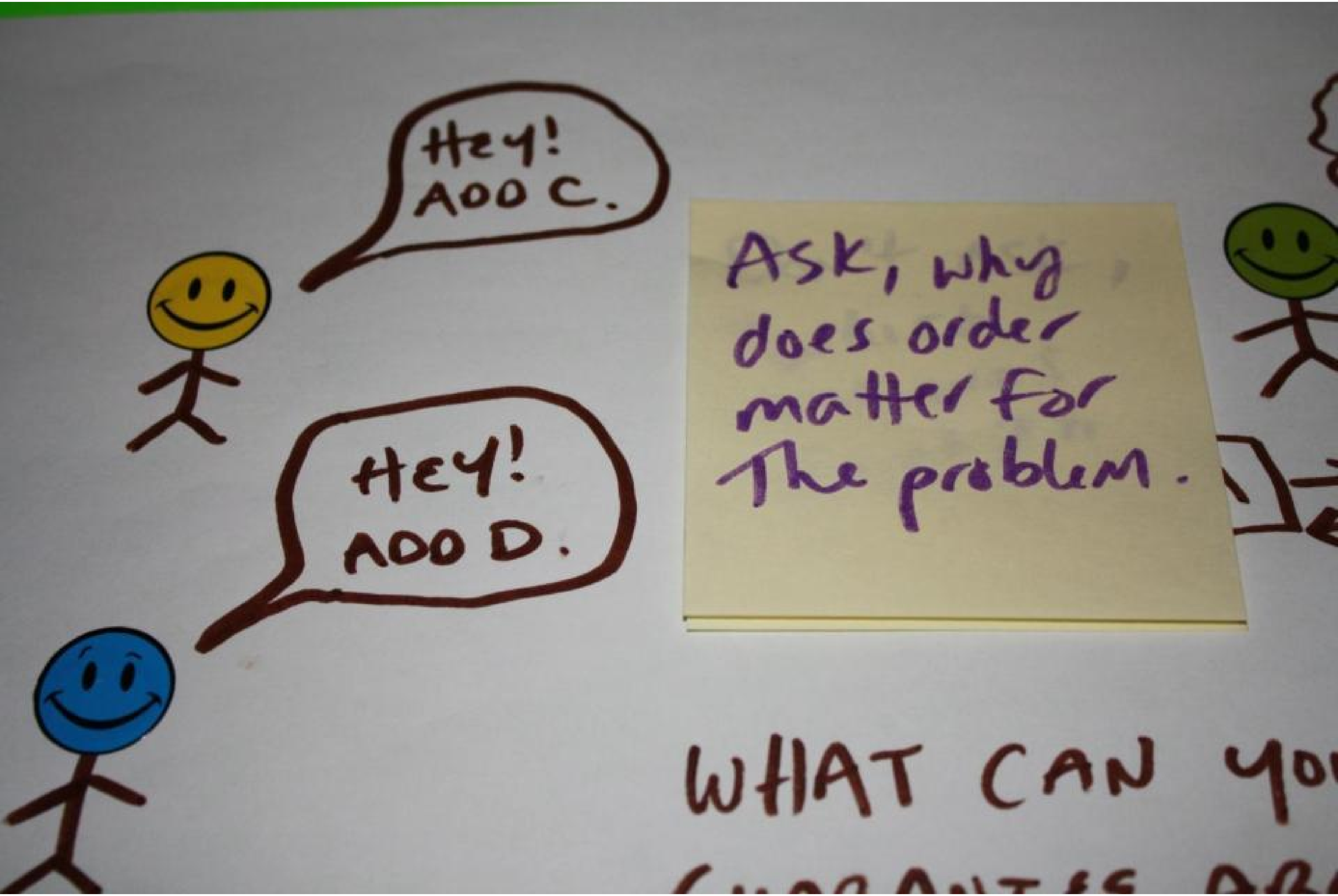

Problem:
Concurrent
insert op
What does
it mean?

Problem:
Concurrent
insert op

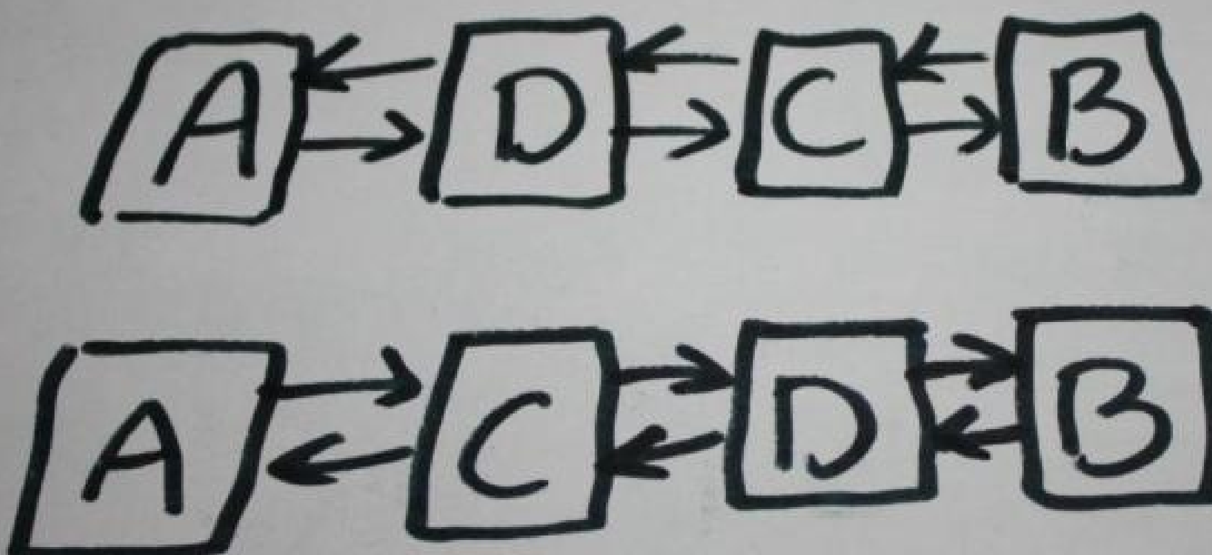
can't
define data
before even
knowing the
problem

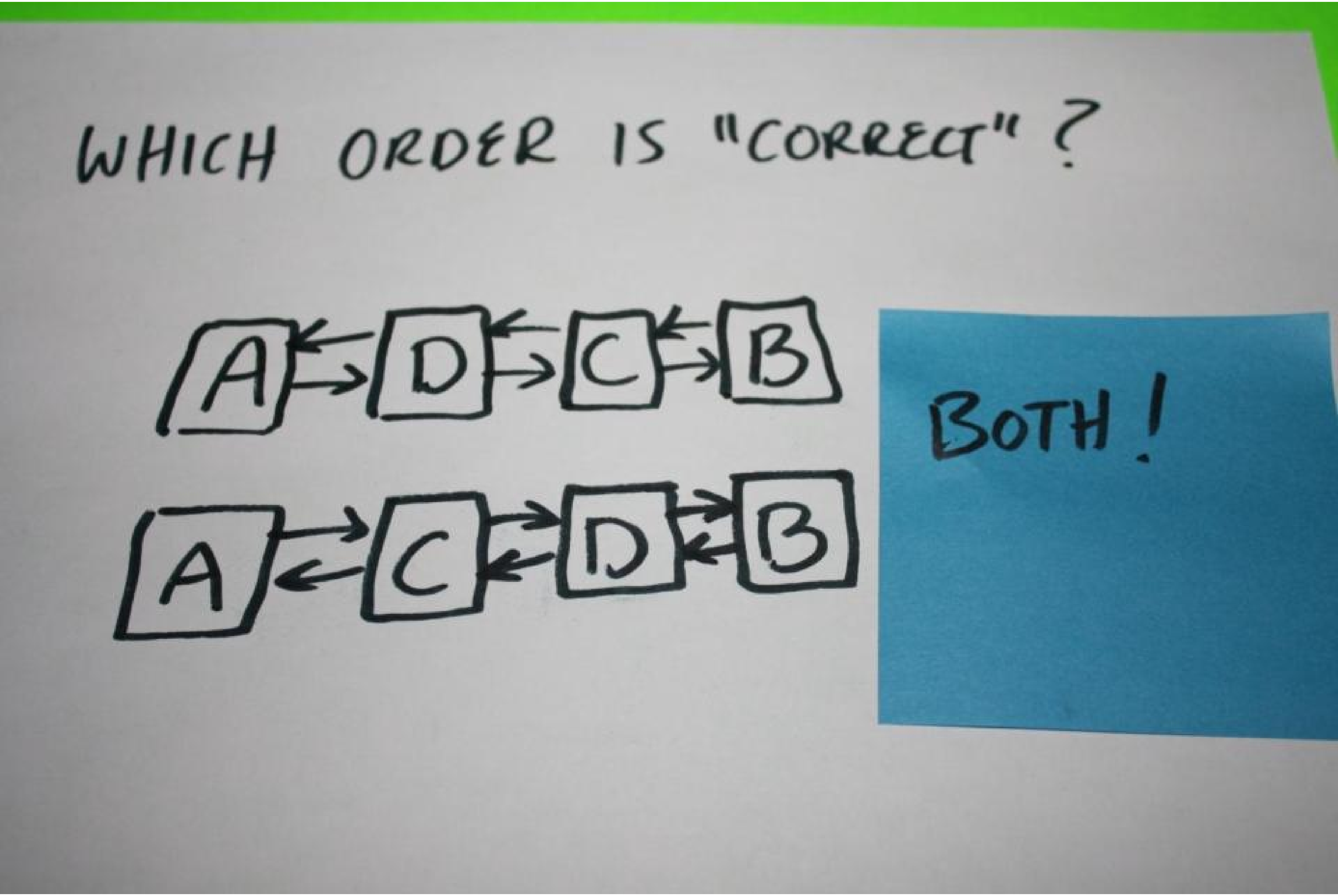

Hey!
ADD C.
Hey!
ADD D.
??
A
B
WHAT CAN YOU GUARANTEE ABOUT THE ORDER HERE?

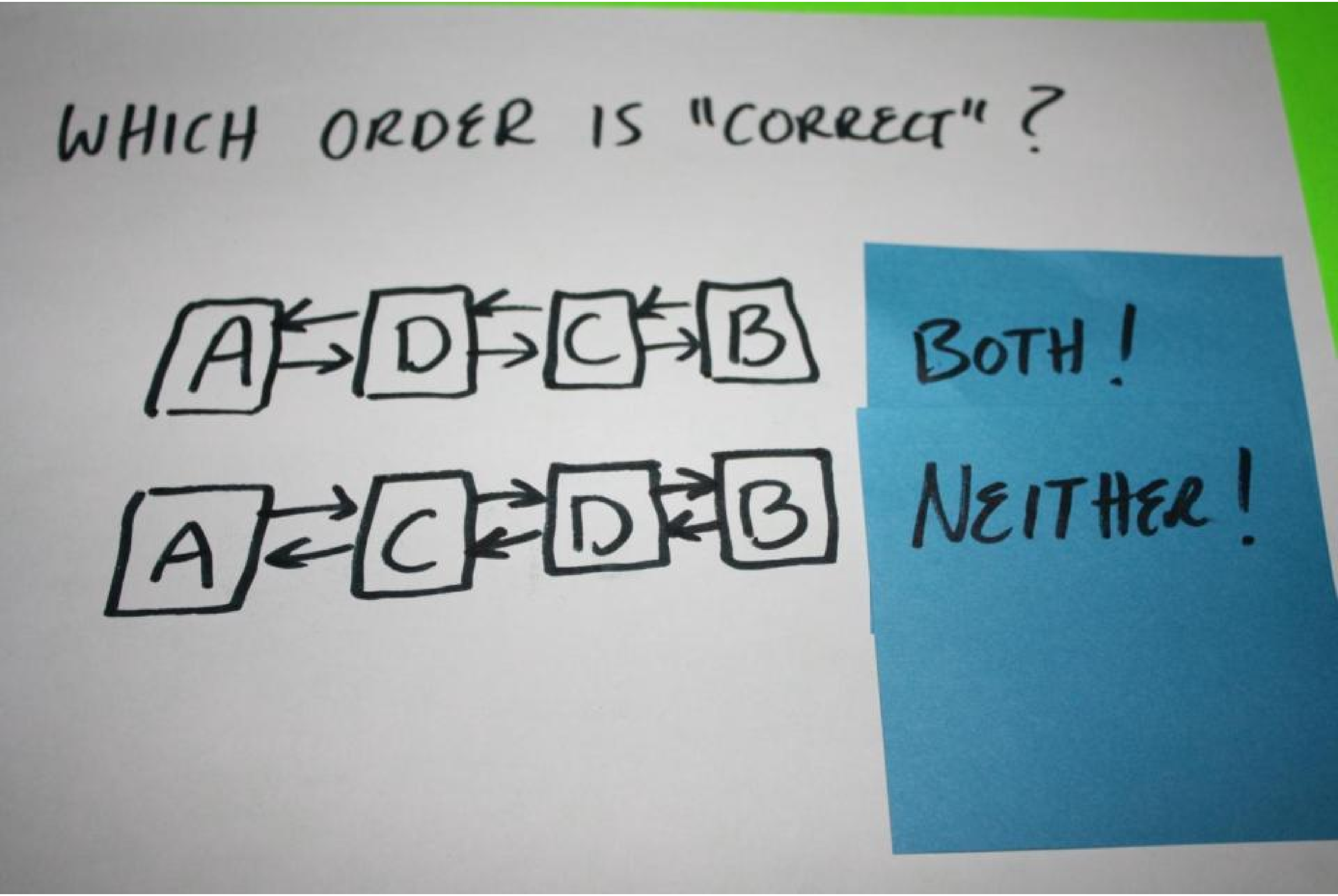

Hey! ADD C.
Hey! ADD D.
Don't ask, "which one was first?"
WHAT CAN
GUARANTEE


Hey!
ADD C.
Hey!
ADD D.
Ask, why does order matter for The problem.
WHAT CAN

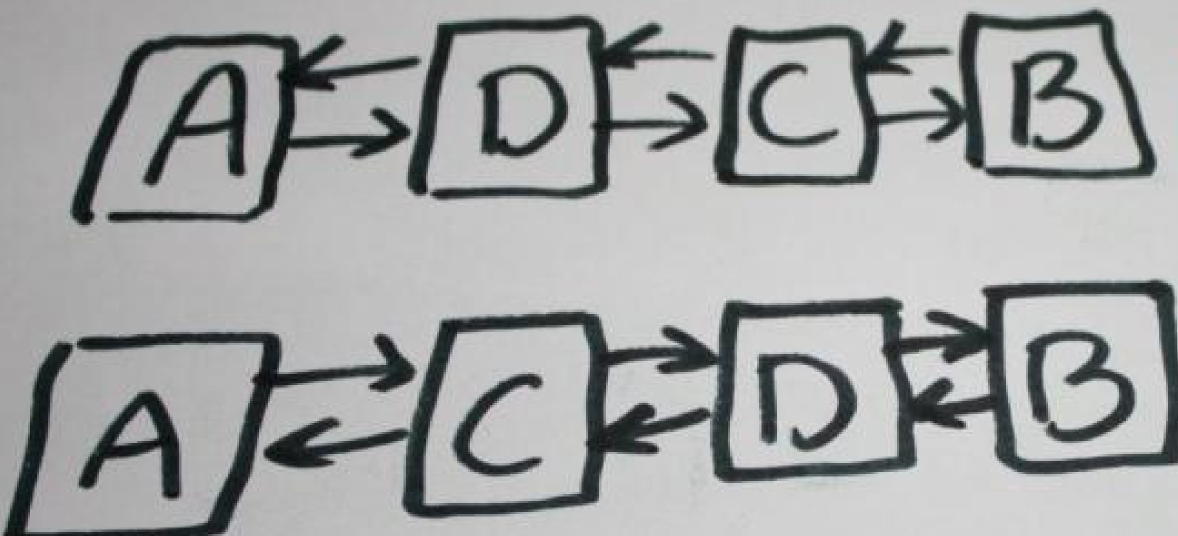
WHICH ORDER IS "CORRECT"?



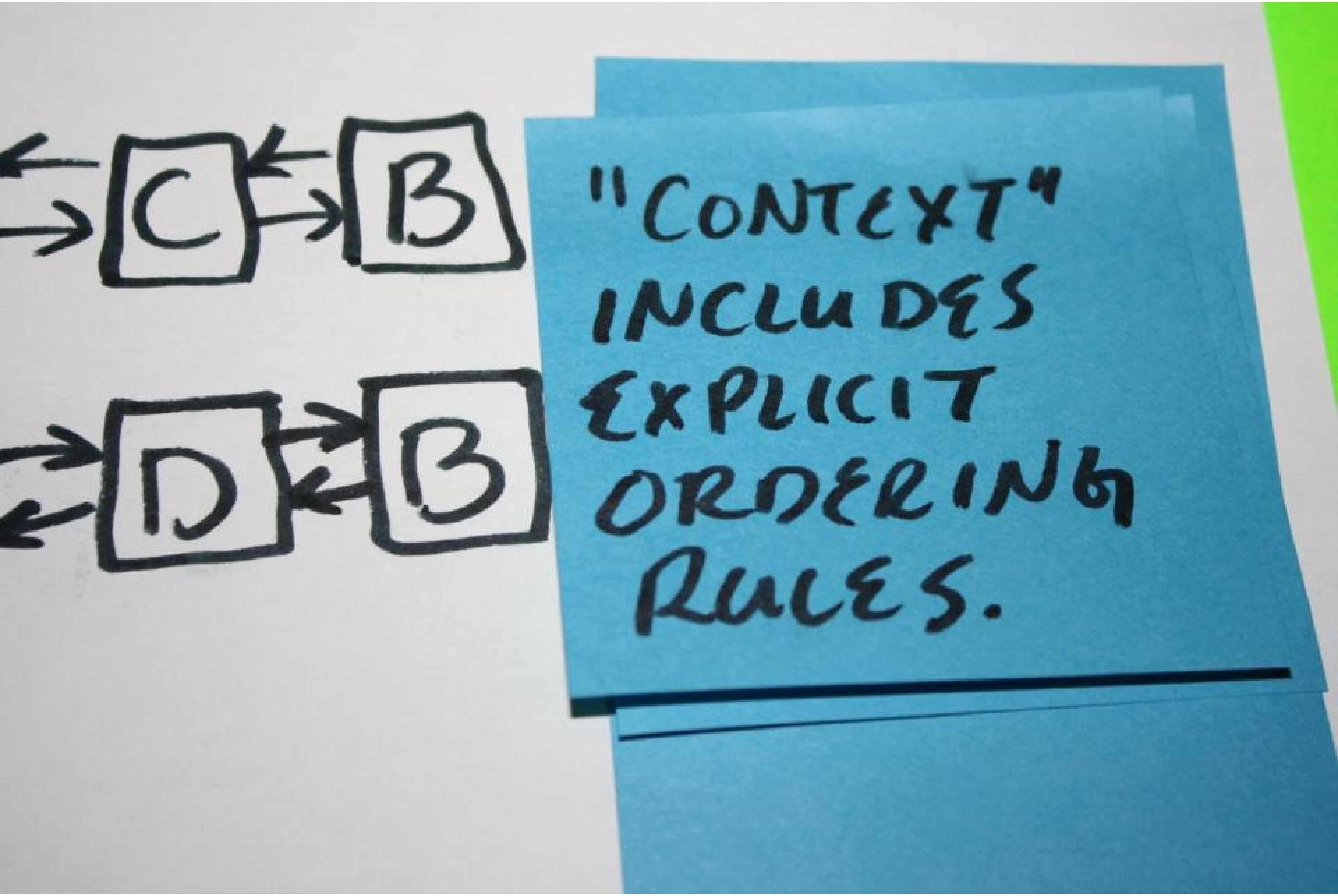

WHICH ORDER IS "CORRECT"?
A
D
C
B
A
C
D
B
BOTH!


WHICH ORDER IS "CORRECT"?
A
D
C
B
A
C
D
B
BOTH!
NEITHER!

WHICH ORDER IS "CORRECT" ?



DEPENDS
ON
CONTEXT.


C
B
D
B
"CONTEXT"
INCLUDES
EXPLICIT
ORDERING
RULES.

Concurrent insert operation needs explicit ordering rule. (Extra dimention of info.)

The data structure would be different to accommodate ordering rule.

Is that all the extra information needed?

Hint: No.

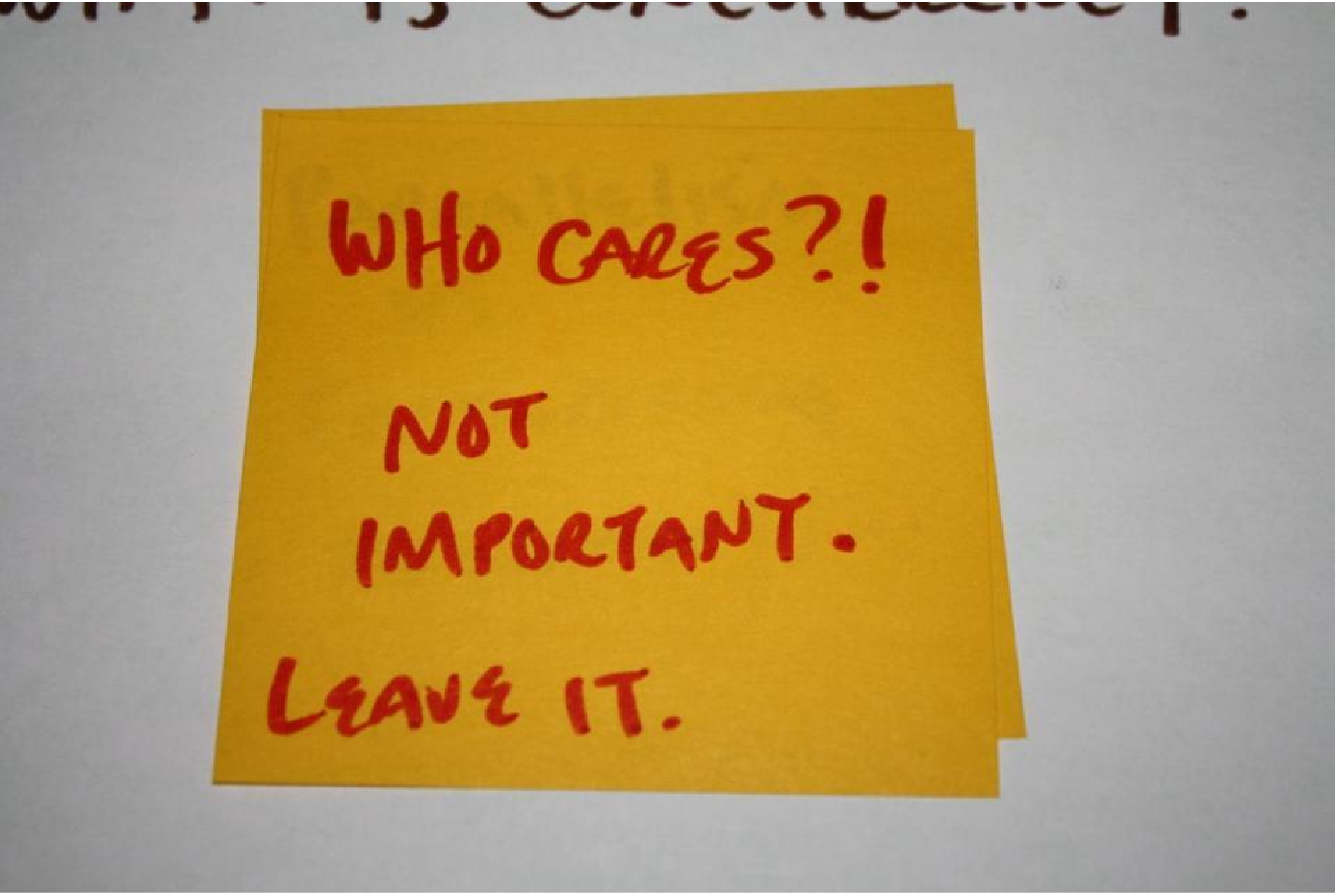
So how would we solve the concurrent problem?

BUT FIRST...

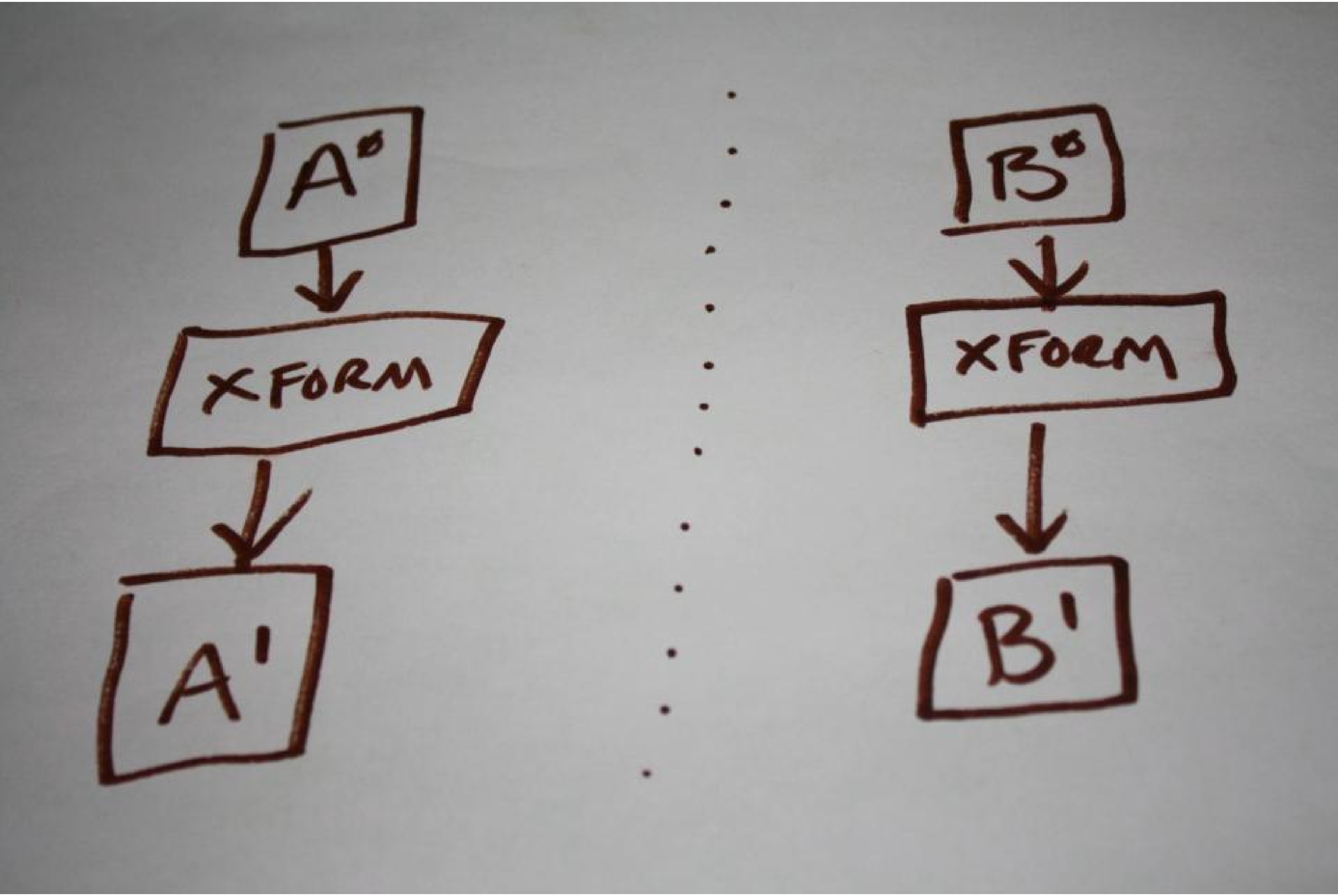
WHAT IS CONCURRENCY?

WHAT IS CONCURRENCY?

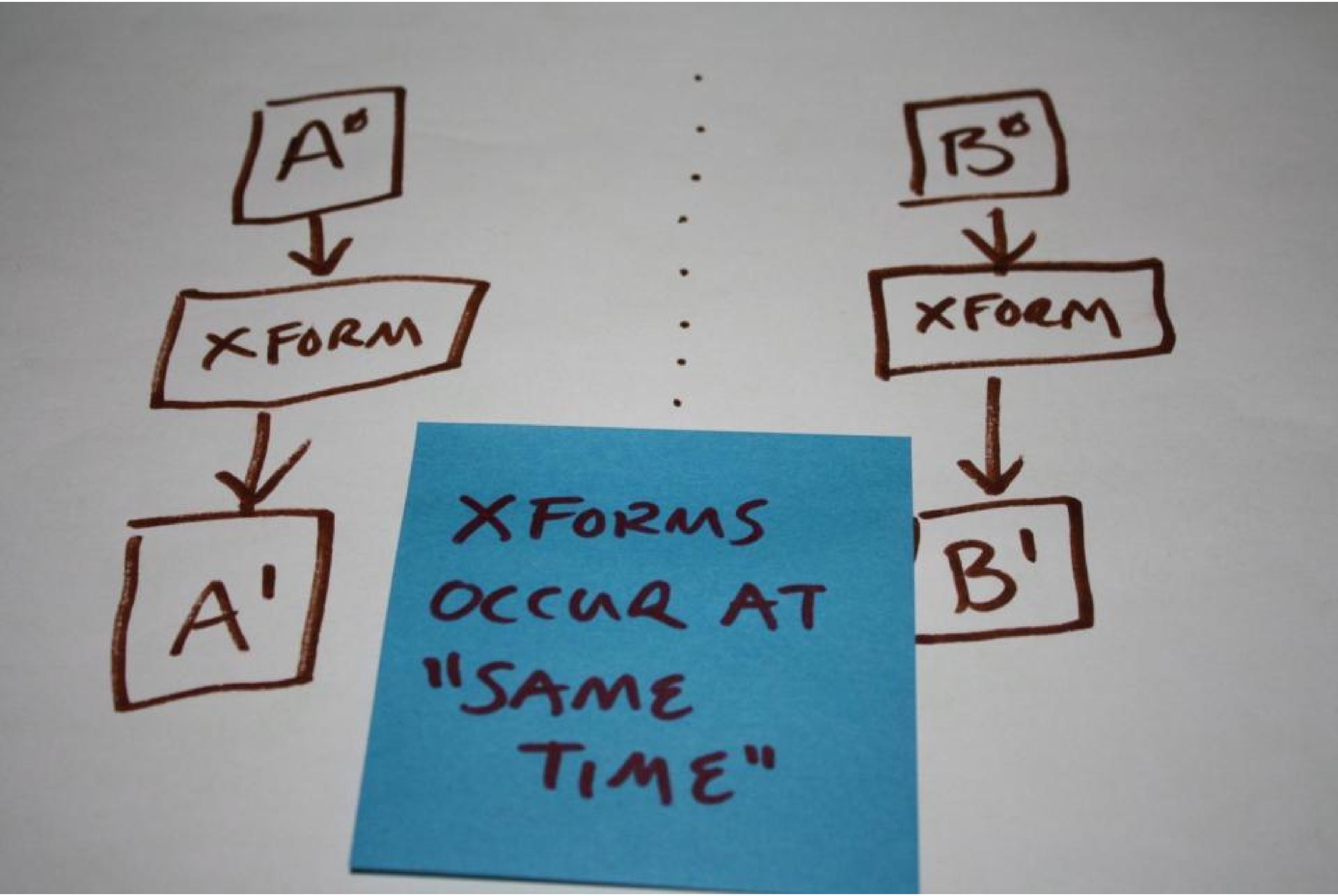
PARRALLELISM
VS.
CONCURRENCY
ARGUMENTS...

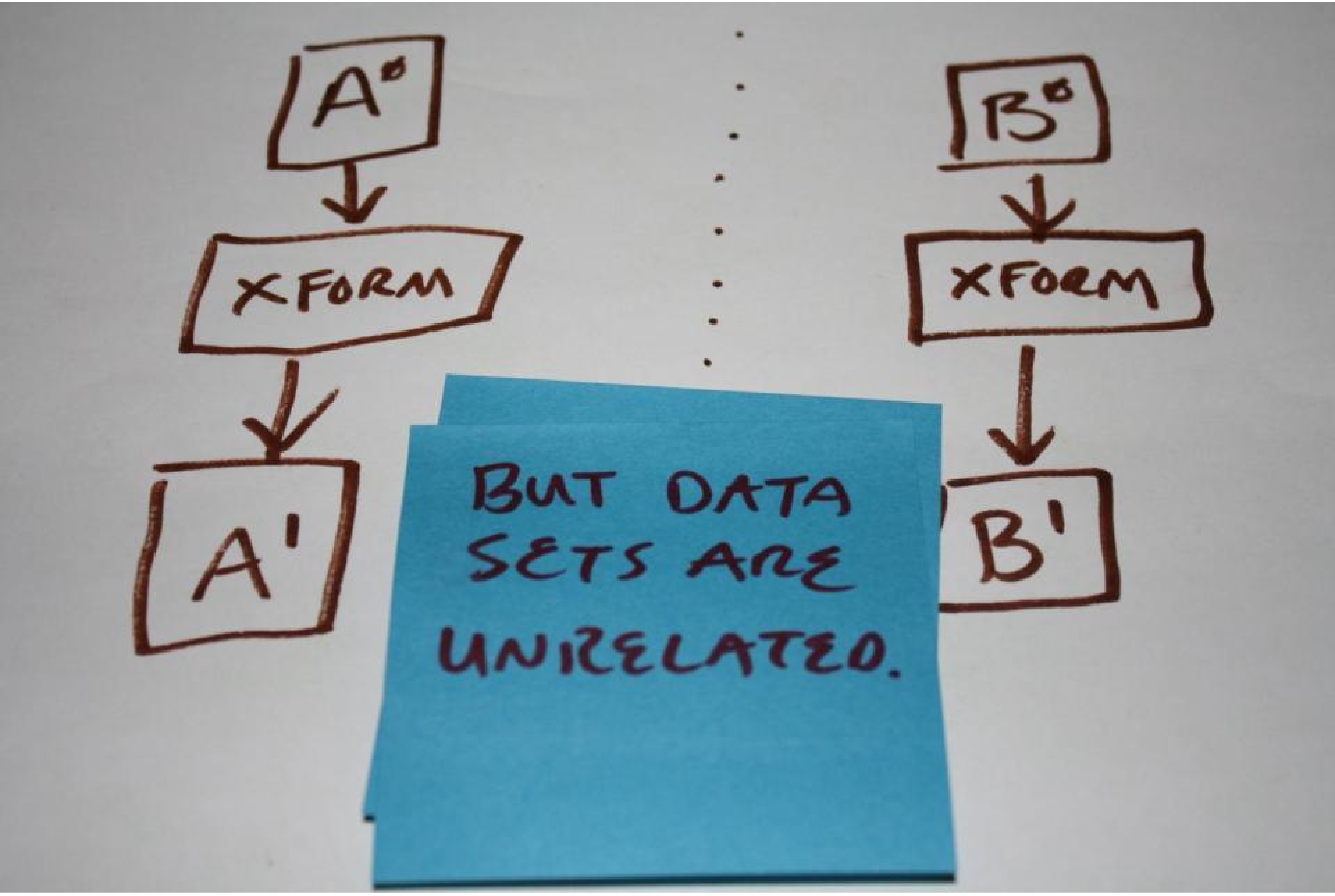

WHO CARES?!
NOT IMPORTANT.
LEAVE IT.

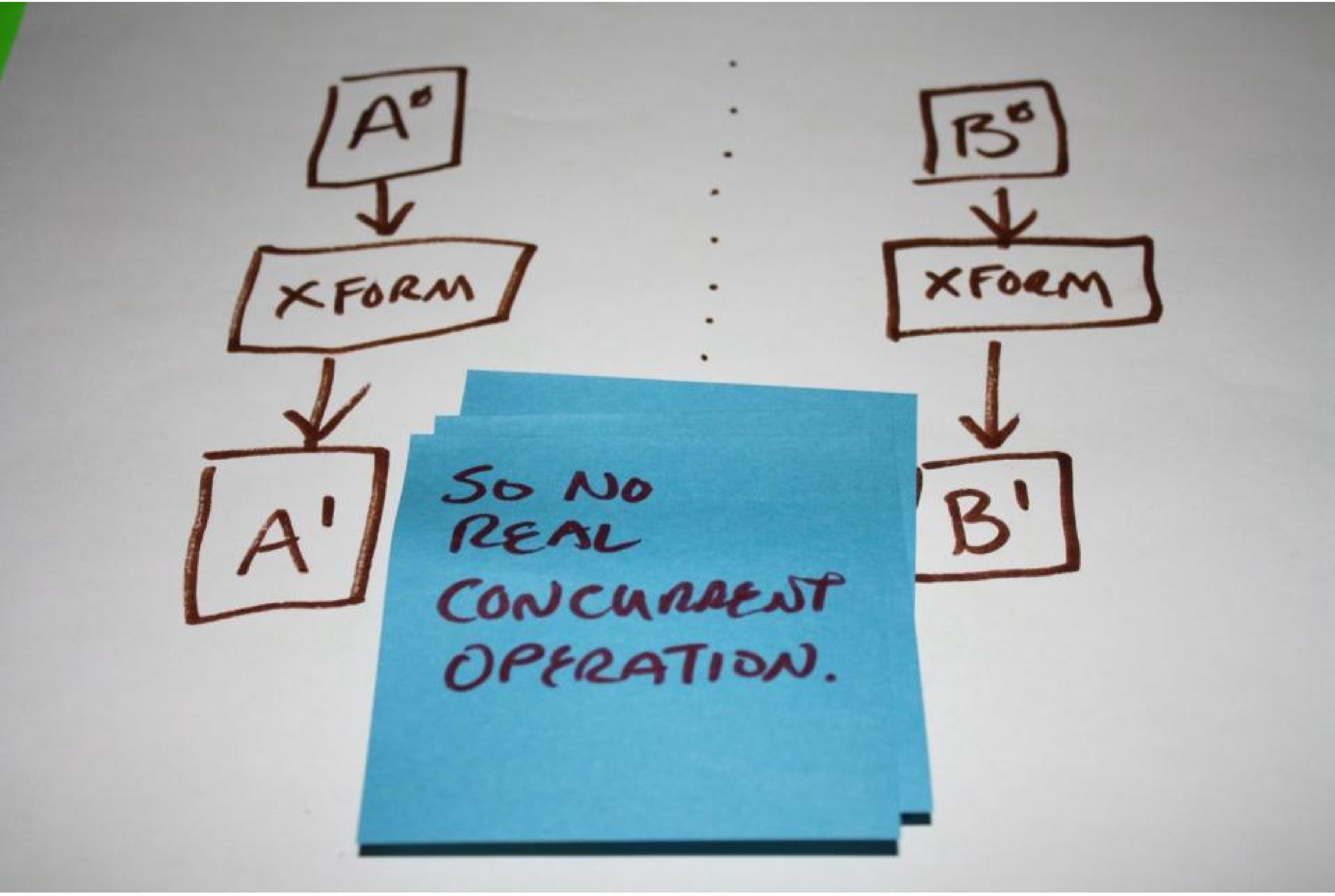
CONCURRENCY IS
TRANSFORMATION OF
SHARED DATA SET.



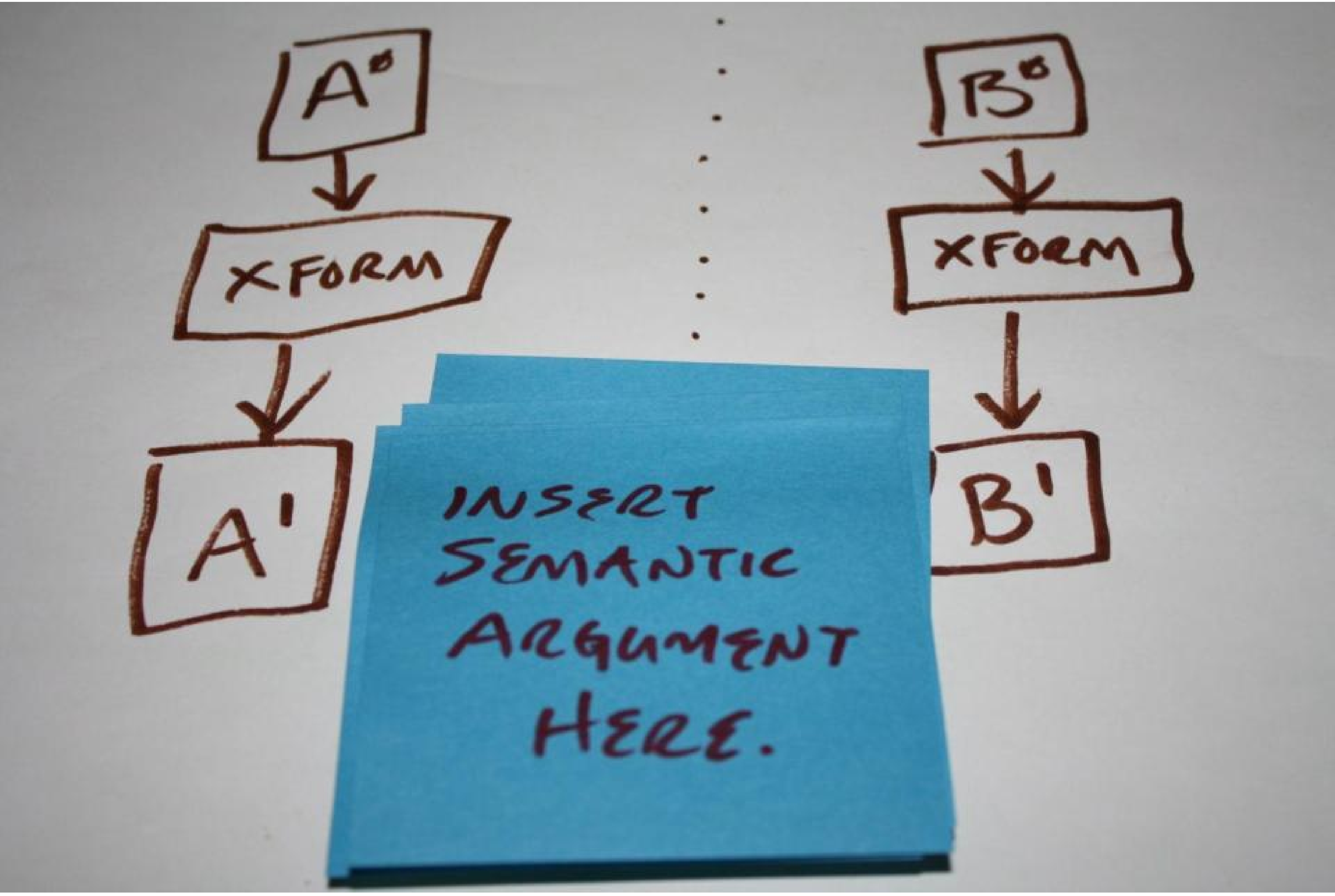
A⁰
XFORM
A'
B⁰
XFORM
B'


A°
XFORM
A'
B°
XFORM
B'
XFORMS OCCUR AT "SAME TIME"

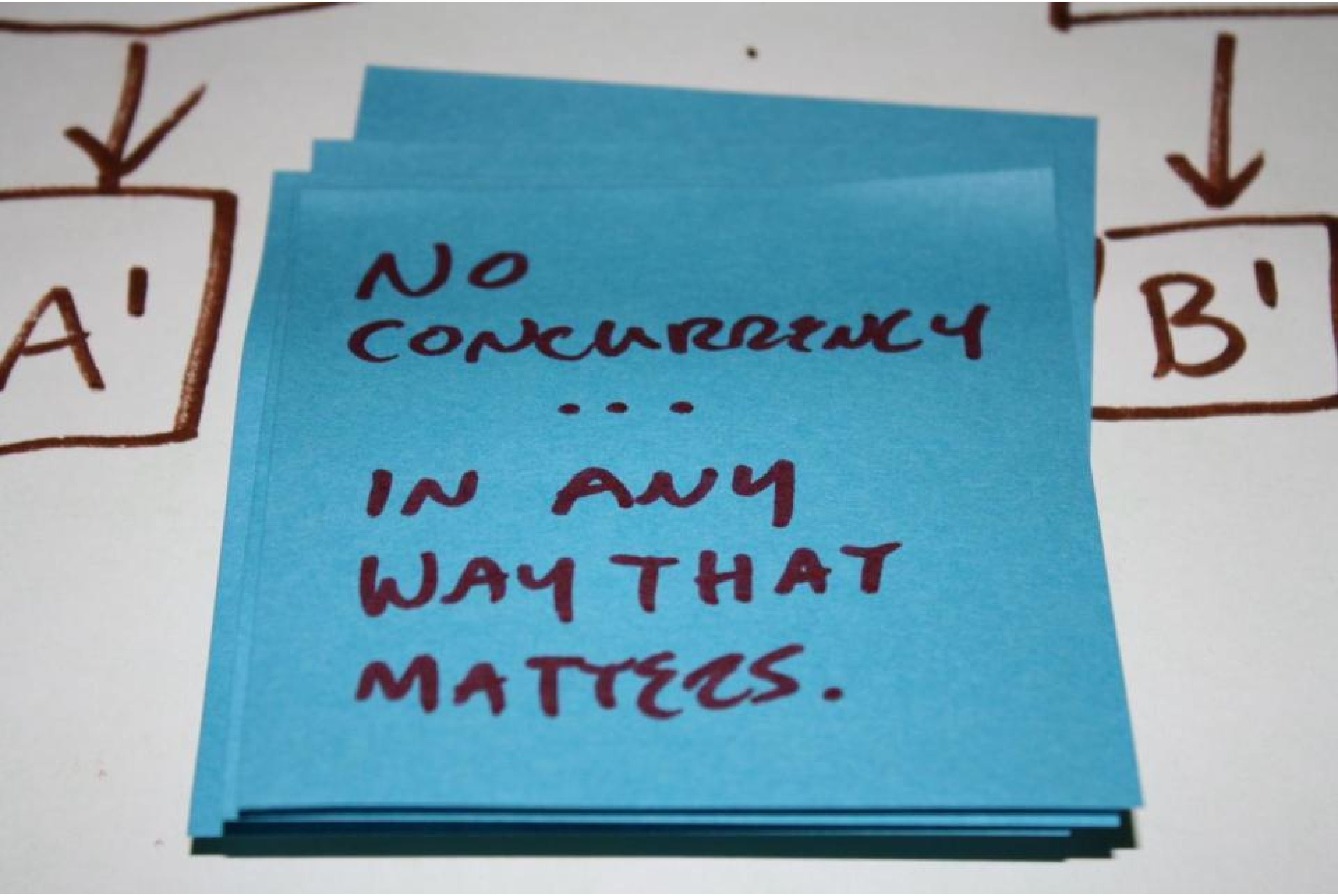

A°
XFORM
A'
B°
XFORM
B'
BUT DATA SETS ARE UNRELATED.

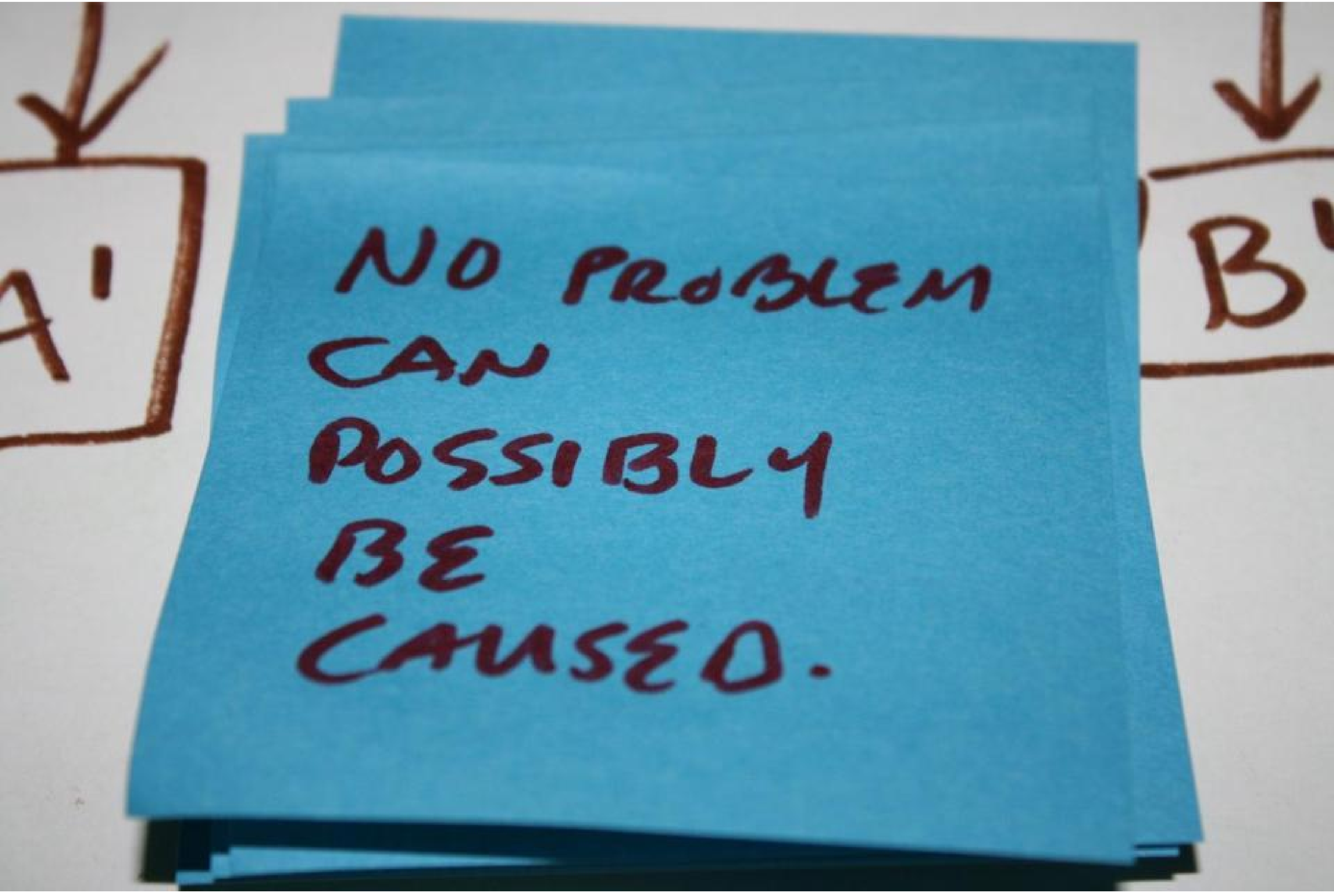


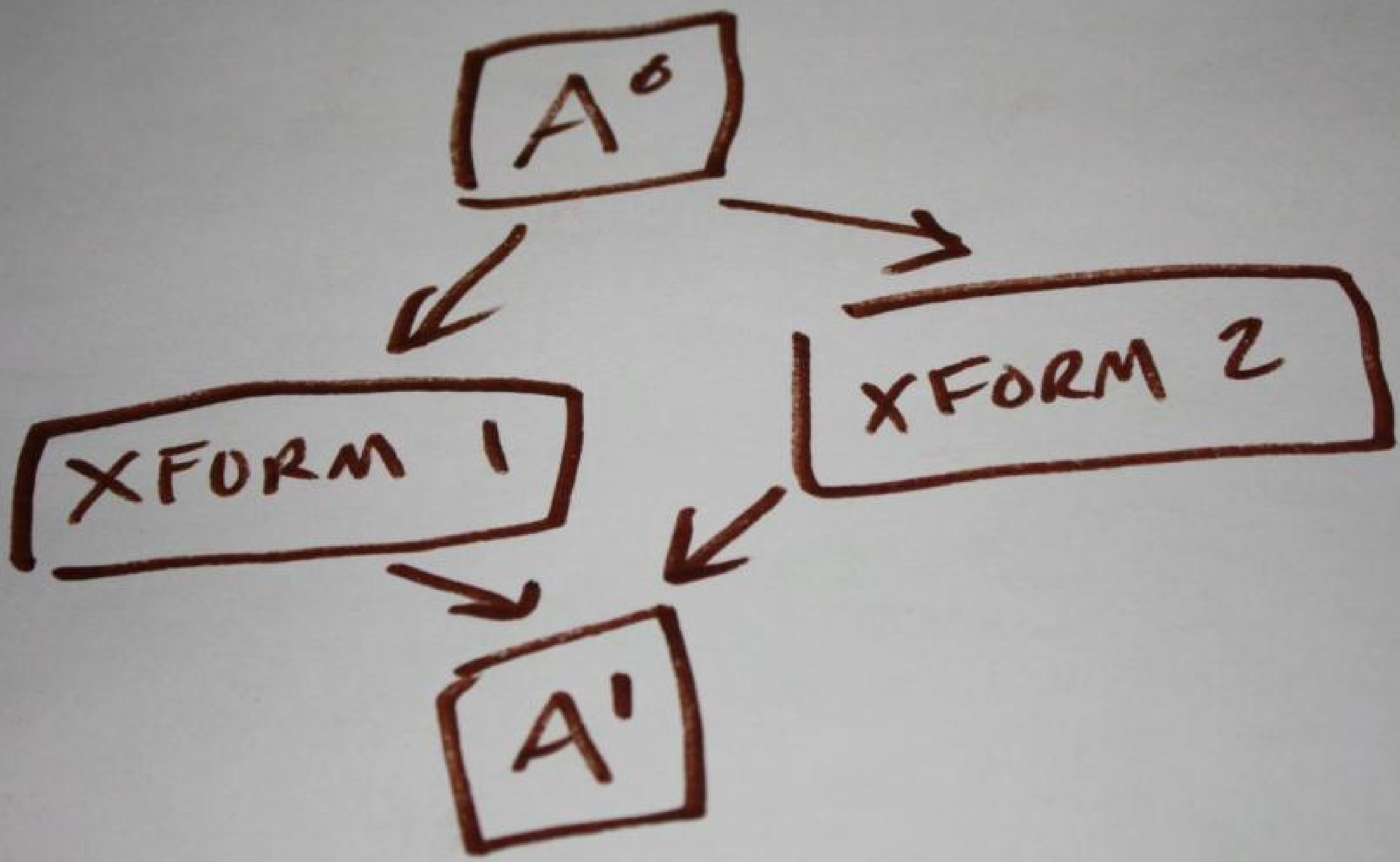
A°
XFORM
A'
B°
XFORM
B'
SO NO REAL CONCURRENT OPERATION.

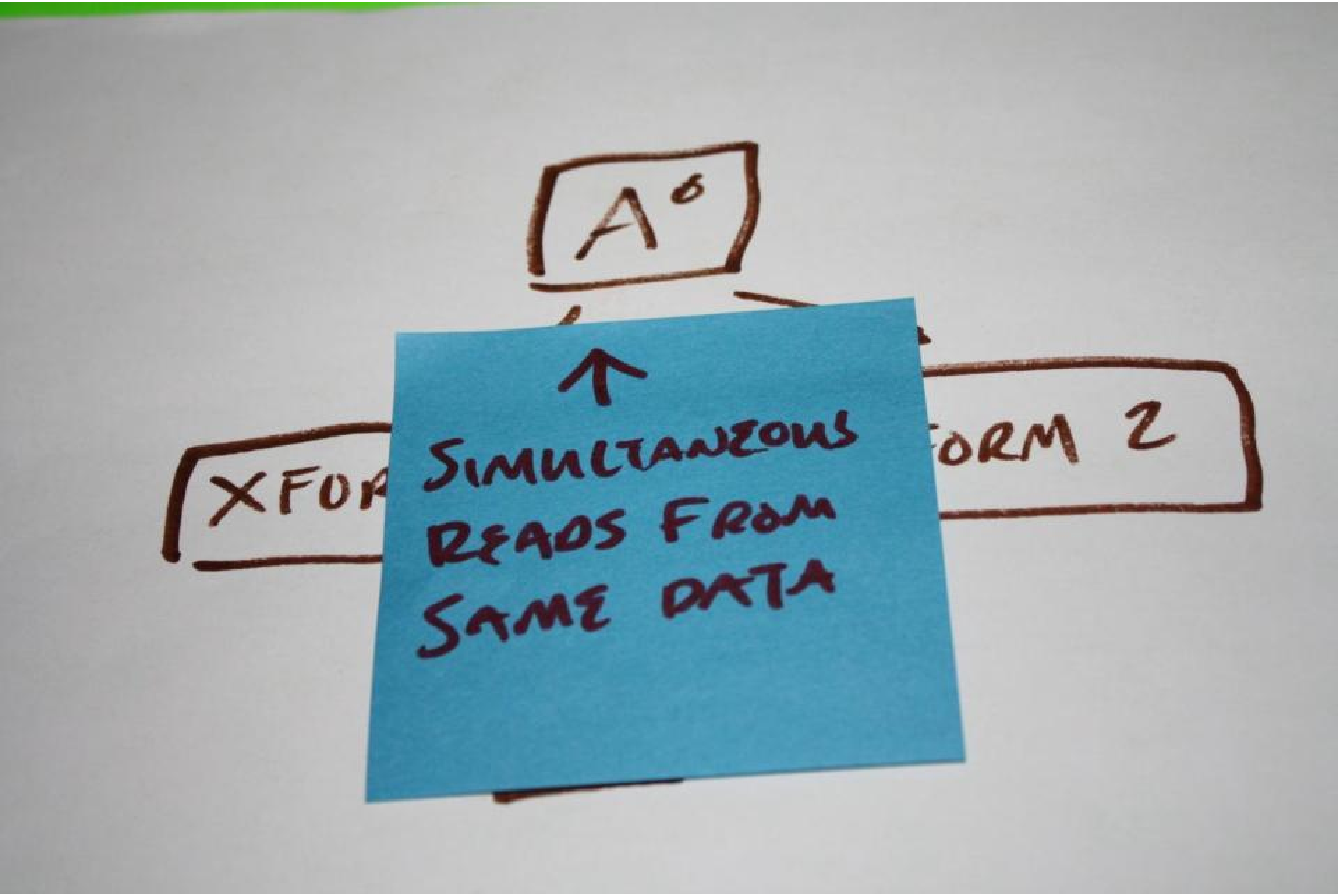


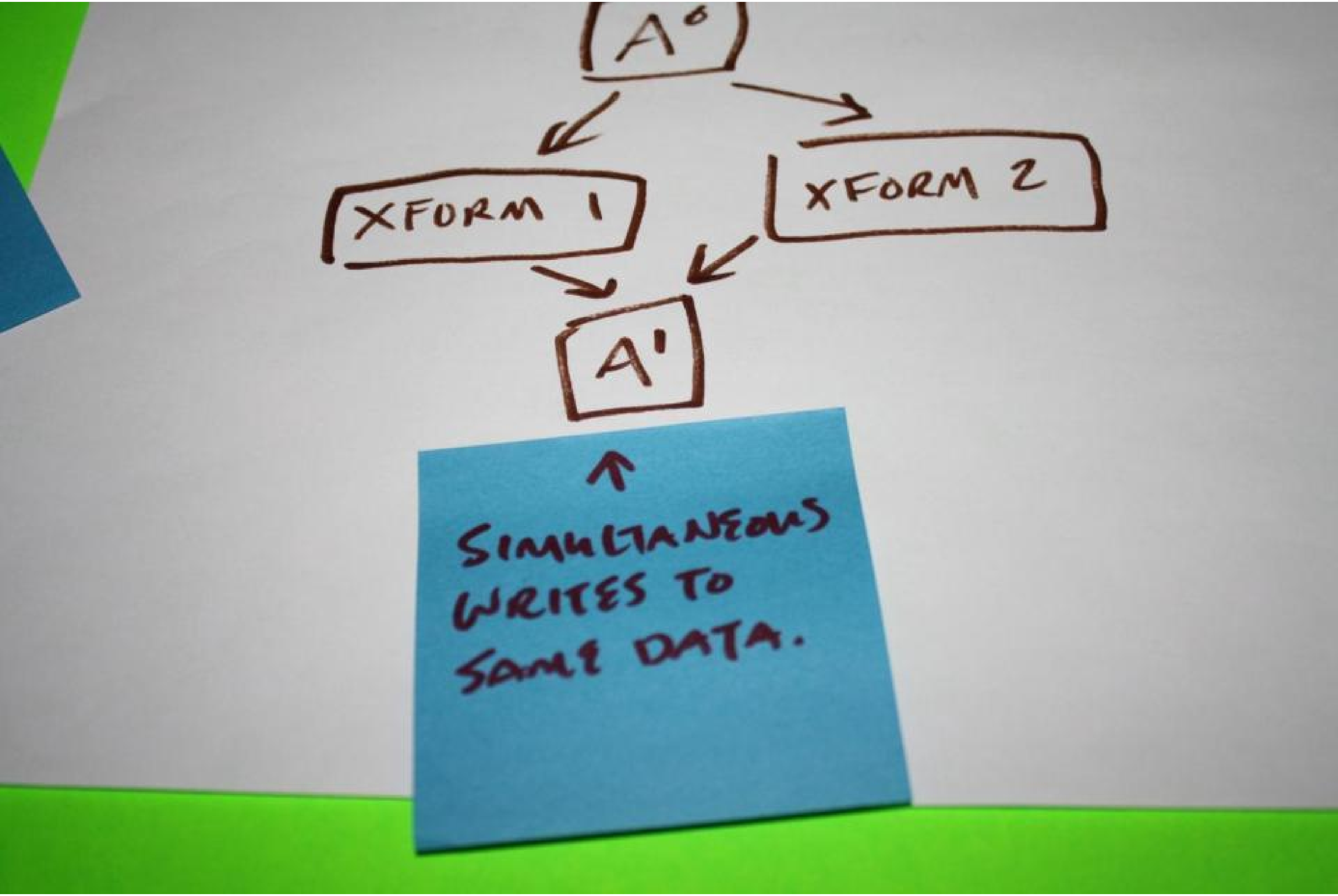
A°
XFORM
A'
B°
XFORM
B'
INSERT SEMANTIC ARGUMENT HERE.

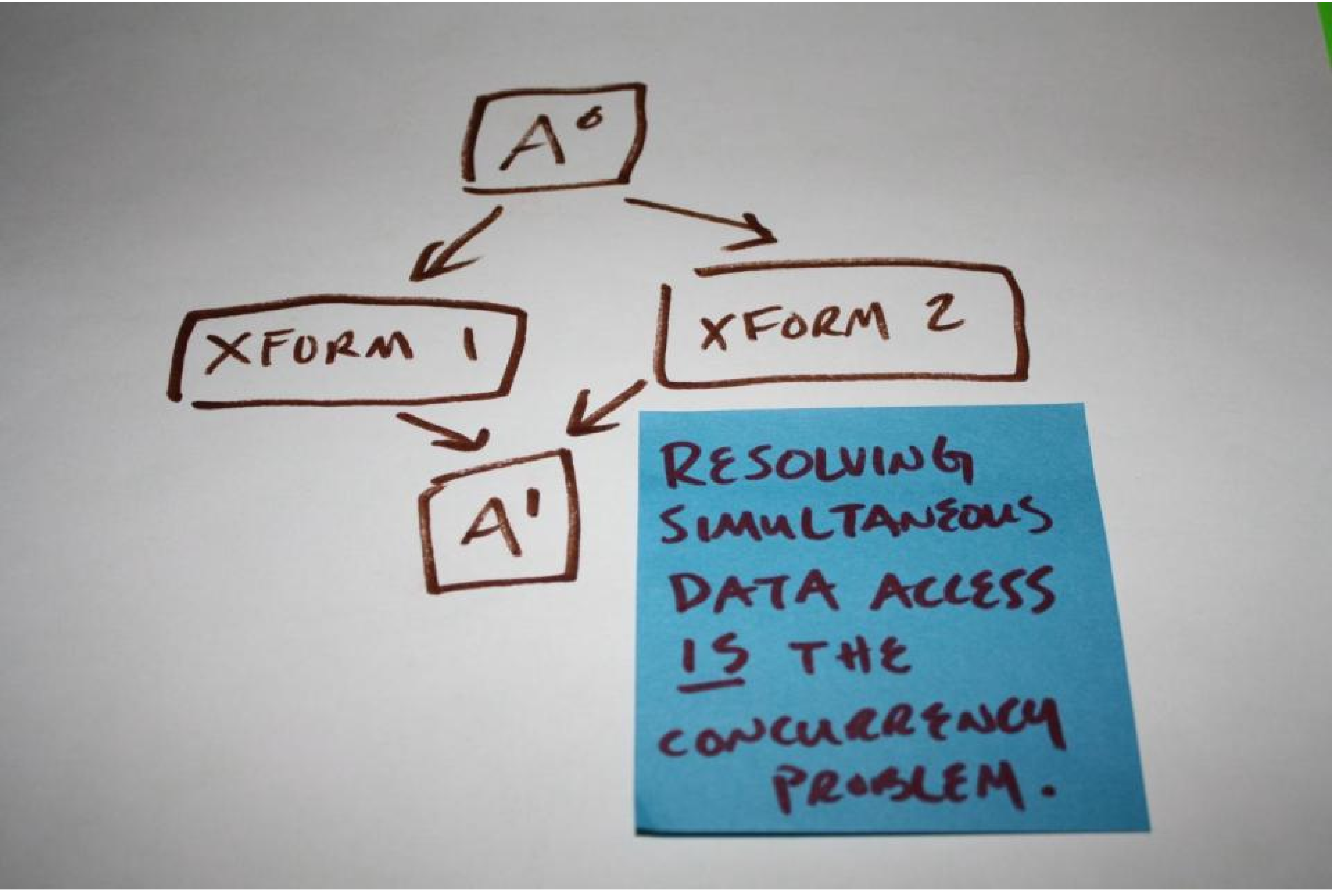

A'
B'
NO CONCURRENCY
...
IN ANY WAY THAT MATTERS.


NO PROBLEM
CAN
POSSIBLY
BE
CAUSED.

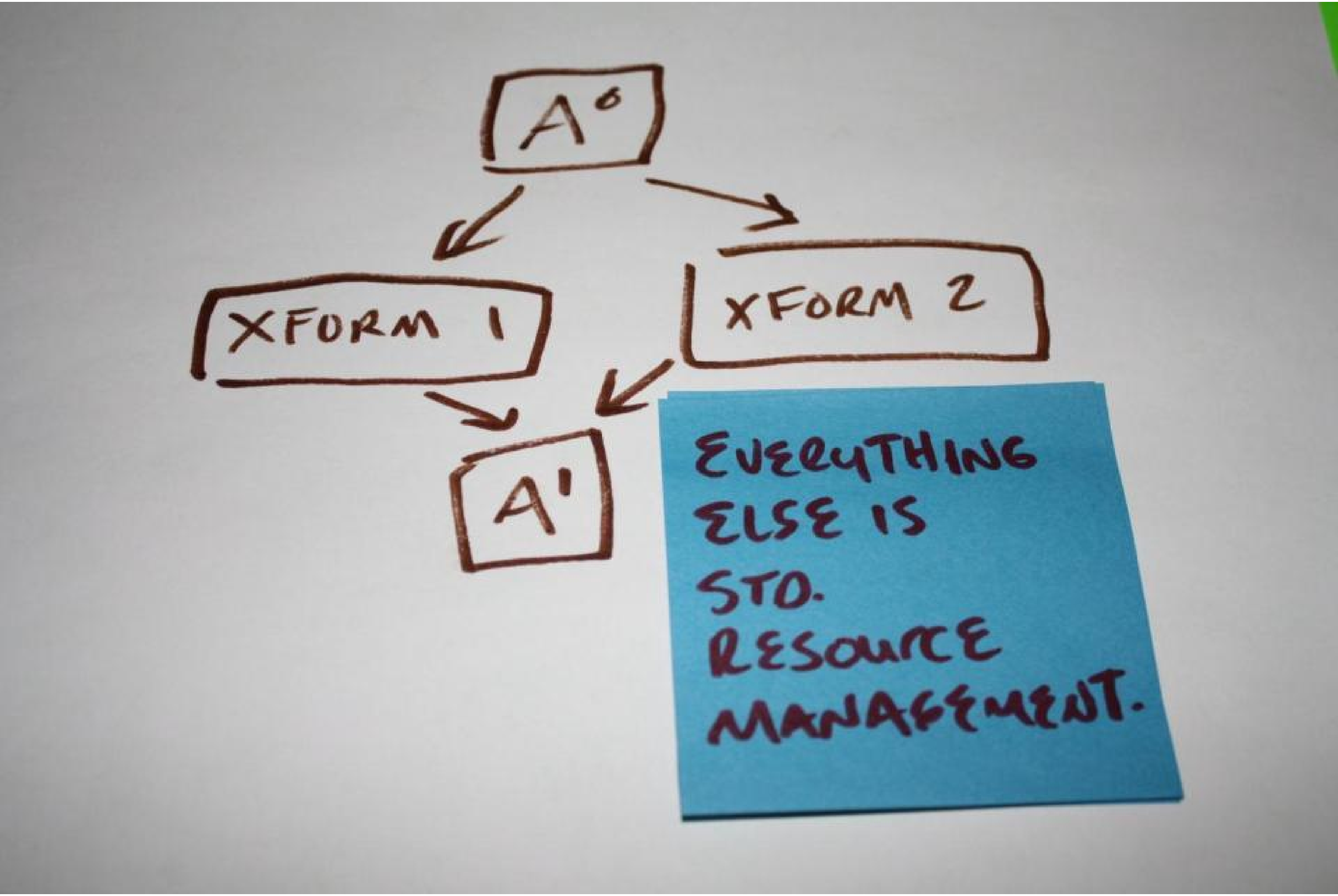

A⁰
XFORM 1
XFORM 2
A'


A°
XFOR
FORM 2
SIMULTANEOUS
READS FROM
SAME DATA

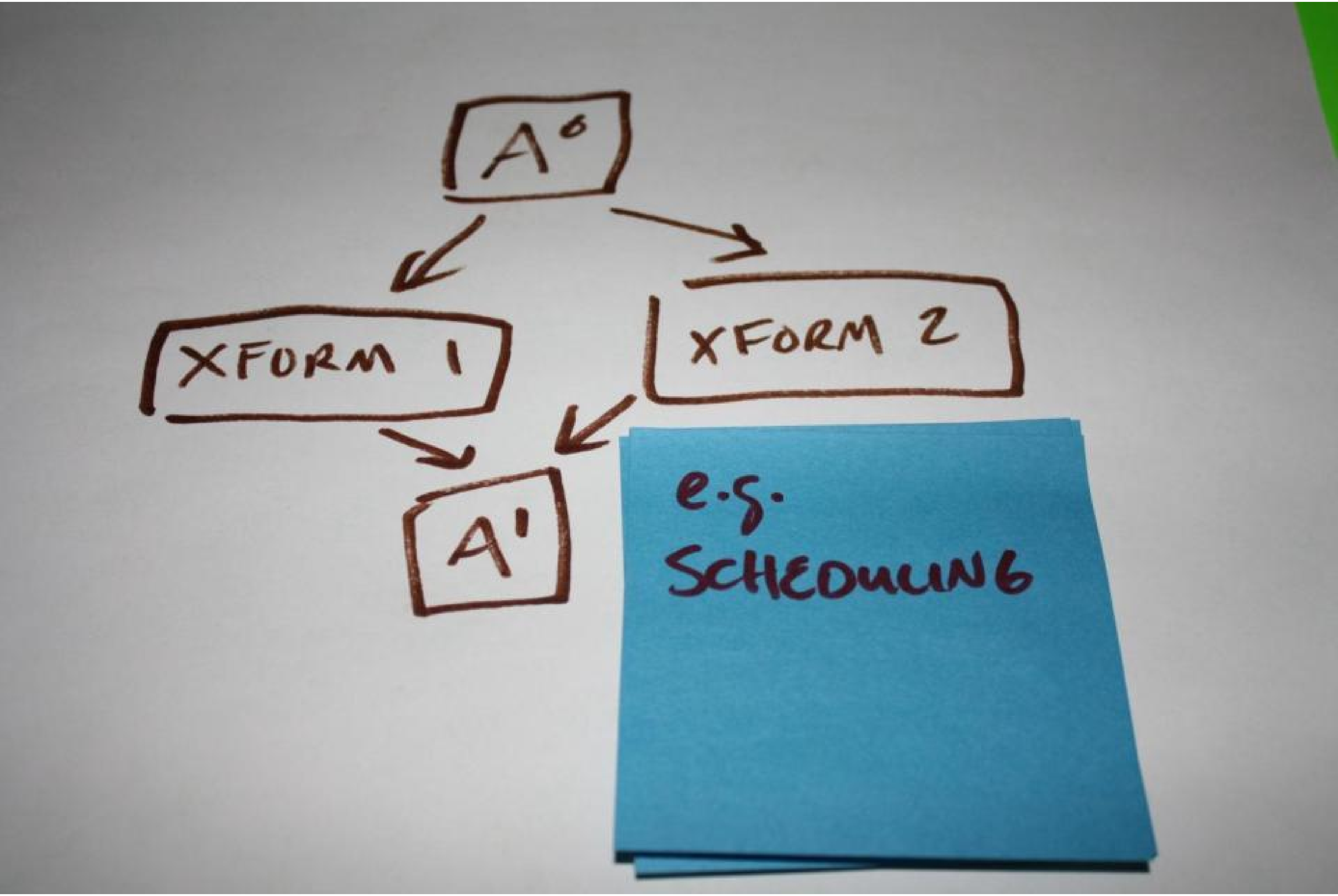

A0
XFORM 1
XFORM 2
A'
SIMULTANEOUS WRITES TO SAME DATA.



A⁰
XFORM 1
XFORM 2
A¹
RESOLVING SIMULTANEOUS DATA ACCESS IS THE CONCURRENCY PROBLEM.



A^0
XFORM 1
XFORM 2
A'
EVERYTHING ELSE IS STD. RESOURCE MANAGEMENT.


A°
XFORM 1
XFORM 2
A'
e.g.
SCHEDULING

Concurrent data would be divided in to shared and unshared data for xforms.

- Doubly-linked list makes no such distinction.
- All sequential data structs presume all shared.

Concurrent data would be divided by readers and writers of data.

- Doubly-linked list makes no such distinction.
- All sequential data structs presume anywhere read/write.

Look at any level of parallelism to see shared data for transforms.

FIRST - VARIOUS PARALLELISM

- INSTRUCTION LEVEL
- MULTI - THREADING
- MULTI - CORE, SHARED MEM
- MULT - CORE, INDEPENDENT MEM
- MULT - MACHINES

WHAT DO ALL OF
THESE HAVE
IN COMMON?

SIMULTANEOUS
XFORM OF
SAME DATA FILE

Note:

Data file just generic term for organized data.
e.g.

- Registers
- Cache (lines)
- Main memory
- ...or actual file on disk.

Concurrency is not a system-wide property

Doubly-linked list data struct assumes all operations follow the same (sequential) rules.

CONCURRENCY IS
AN ATTRIBUTE OF
AN OPERATION.

CONCURRENCY IS AN ATTRIBUTE OF AN OPERATION.

WHICH OPERATION?

Every concurrent operation must have explicitly defined rules.

Data is designed that satisfies all the rules.

But sometimes, attempts are made to use "sequential rules"

For example...

WHAT ABOUT USING
TIMESTAMPS TO CONTROL
ORDER?

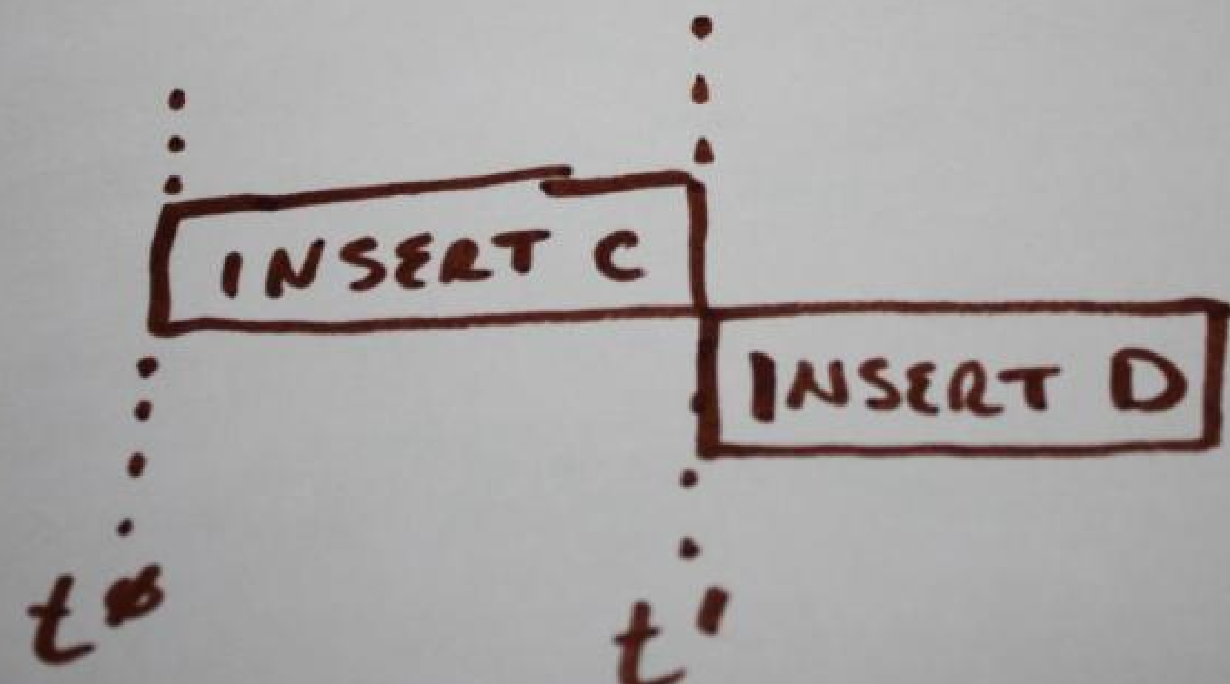
WHAT ABOUT USING TIMESTAMPS TO CONTROL ORDER?

#1 NEED Perfectly Sync'd clocks...

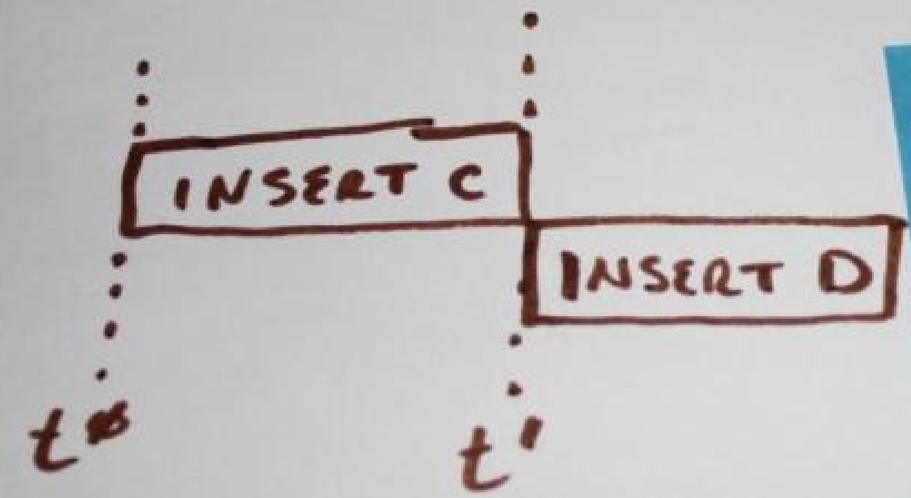
WHAT ABOUT USING TIMESTAMPS TO CONTROL ORDER?

... which are infinitely accurate

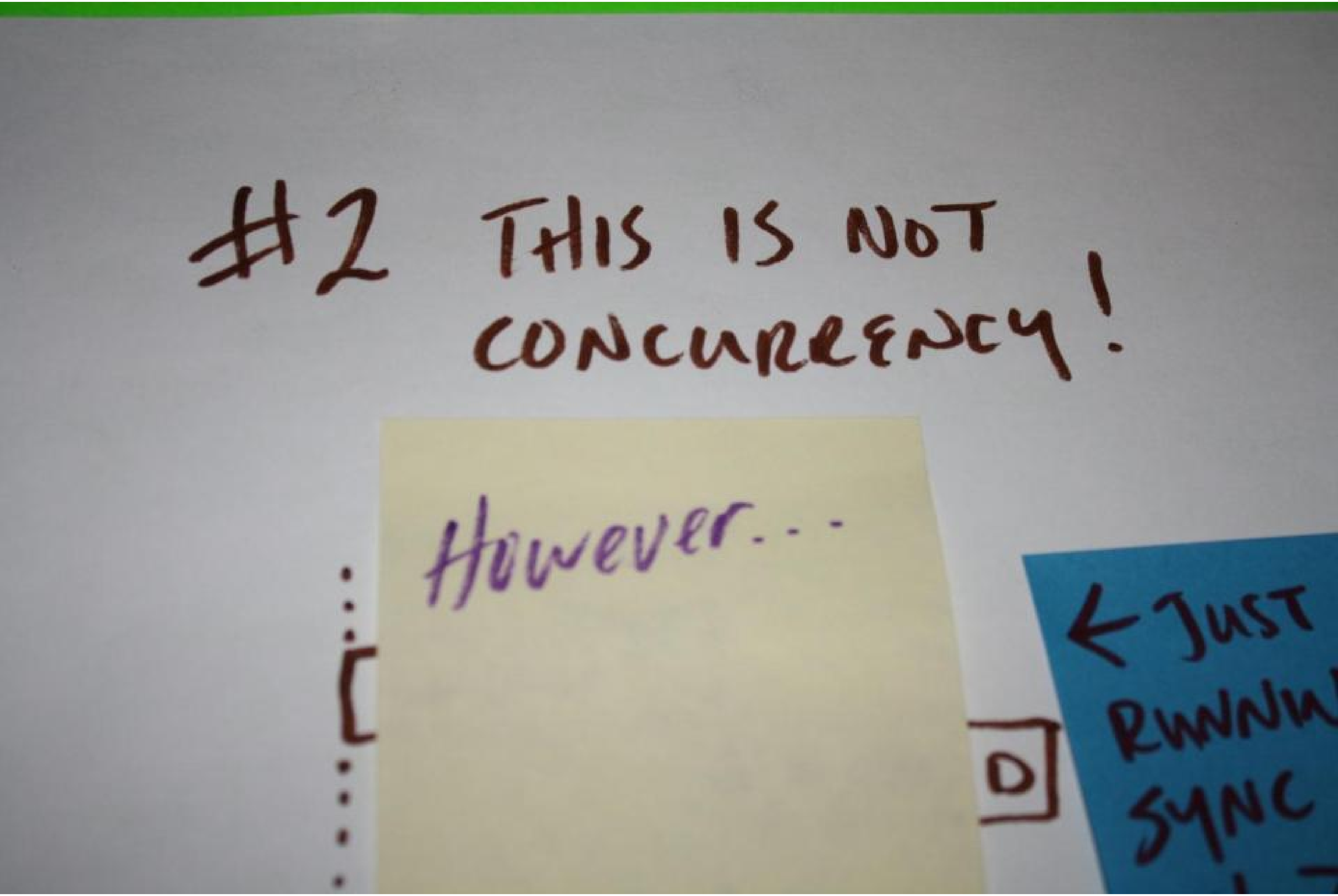
#2 THIS IS NOT CONCURRENCY!



#2 THIS IS NOT CONCURRENCY!

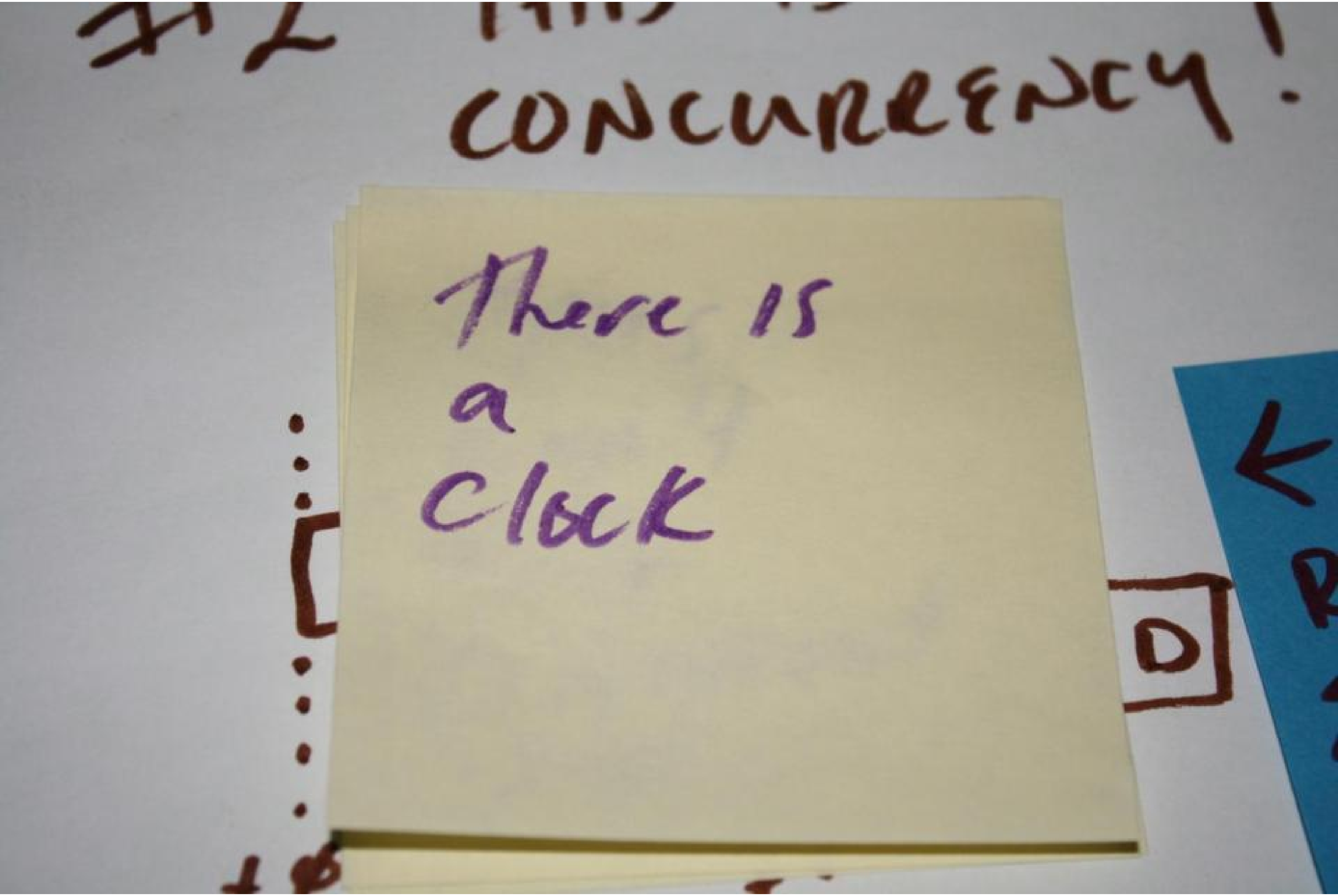


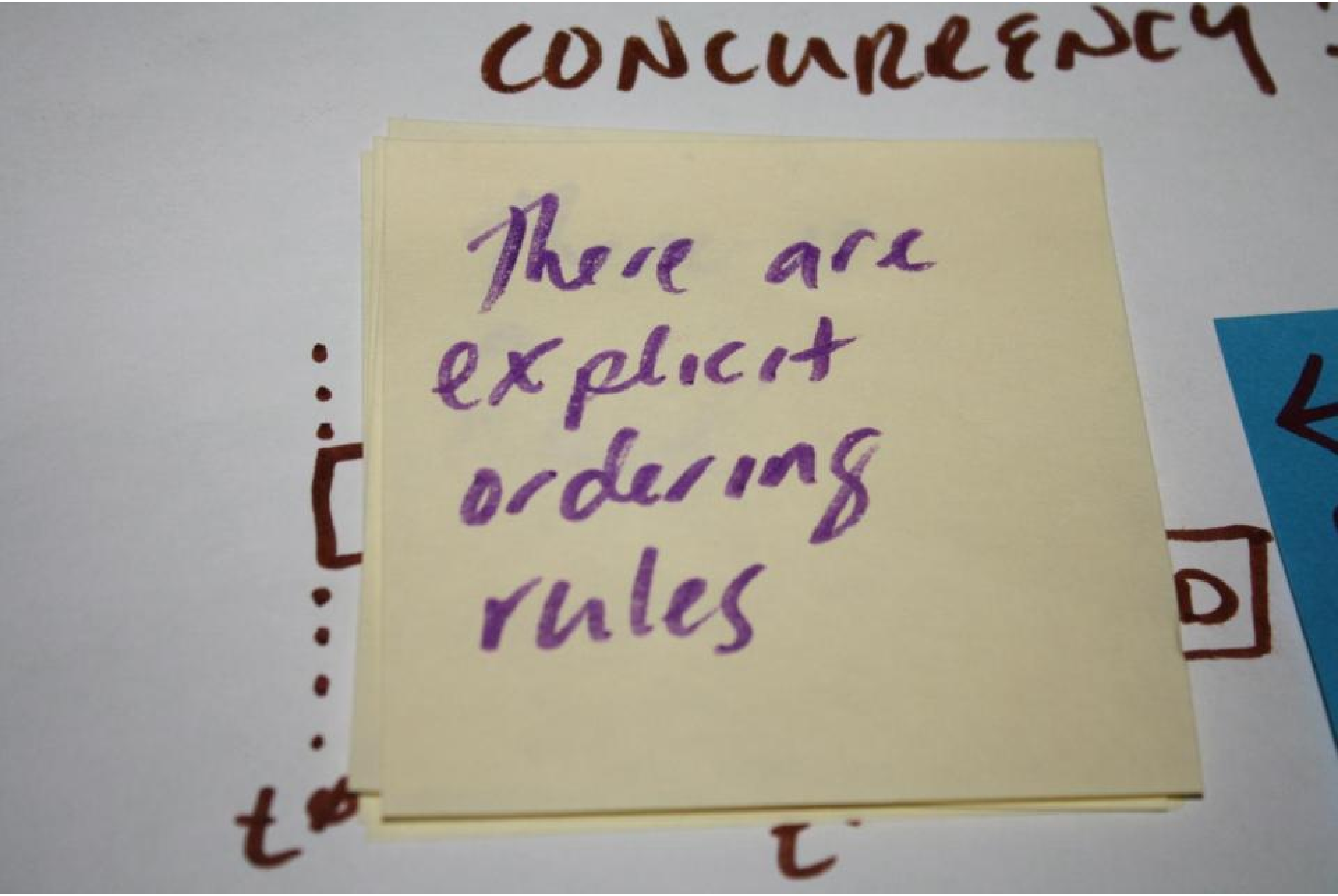
← JUST RUNNING SYNC APP ON >1 THREAD/CORE.


#2 THIS IS NOT CONCURRENCY!
However...
D
← JUST
SYNC

CONCURRENCY!

CPUs
aren't
fully
concurrent
either


CONCURRENCY!
There is
a
clock
D


CONCURRENCY
There are explicit ordering rules
t0

Can't be
more
concurrent
than h/w
allows.

So sometimes using sequential rules work.

And sometimes it's the "right" thing to do.

But it must be done in a well-informed way.

(Know that's what you're doing!)

What needs to be solved per operation?

CONCURRENCY IS <u>FIRST</u> ABOUT RESOLVING DATA SYNCHRONIZATION CONFLICTS.

CONCURRENCY IS **FIRST** ABOUT RESOLVING DATA SYNCHRONIZATION CONFLICTS.

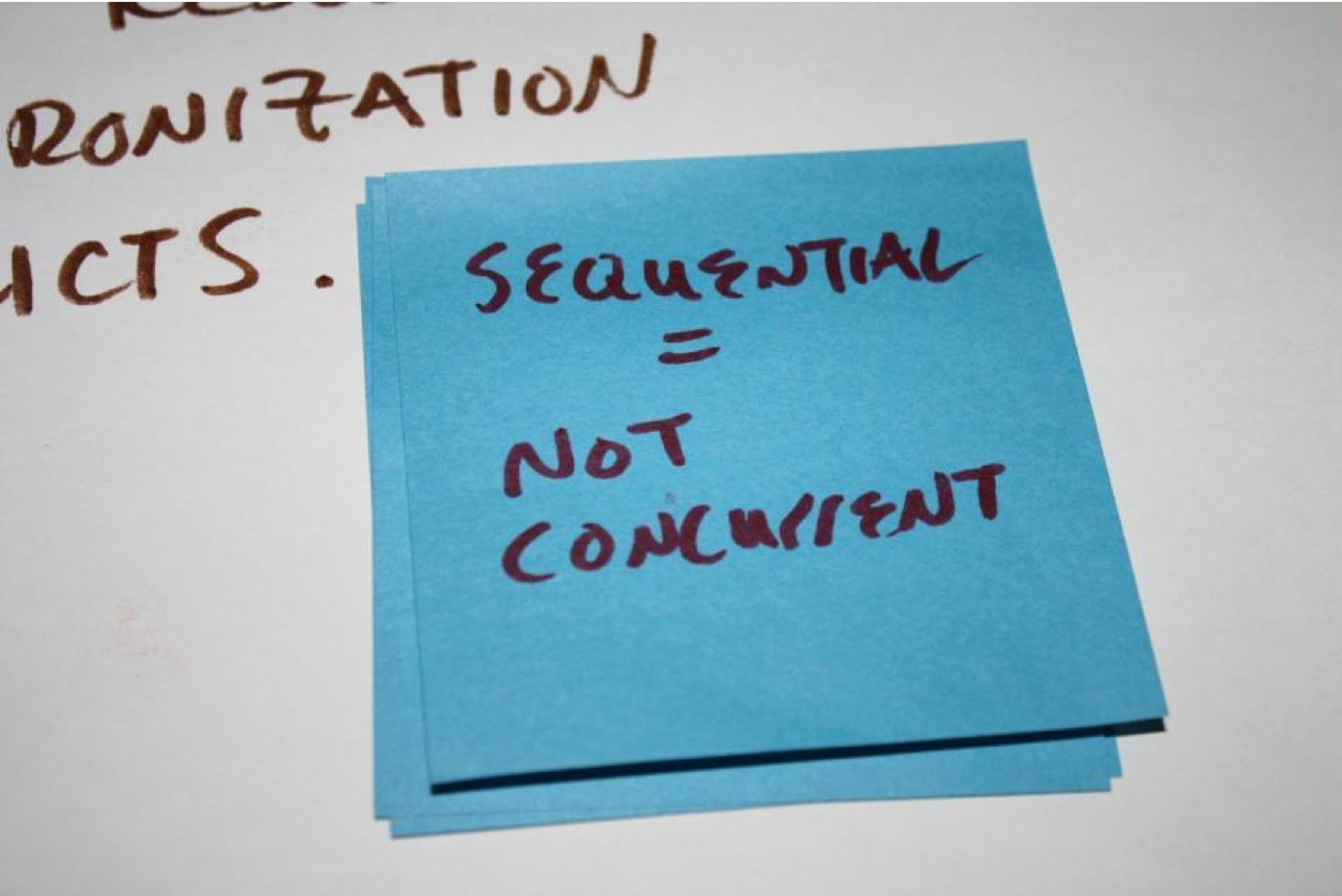
DEFINE DATA IN CONTEXT

CONCURRENCY IS <u>FIRST</u> ABOUT RESOLVING DATA SYNCHRONIZATION CONFLICTS.

MINIMIZE CONFLICTS

CONCURRENCY IS <u>FIRST</u> ABOUT RESOLVING DATA SYNCHRONIZATION CONFLICTS.

CONFLICTS =
SEQUENTIAL
DATA
DEPENDENCY

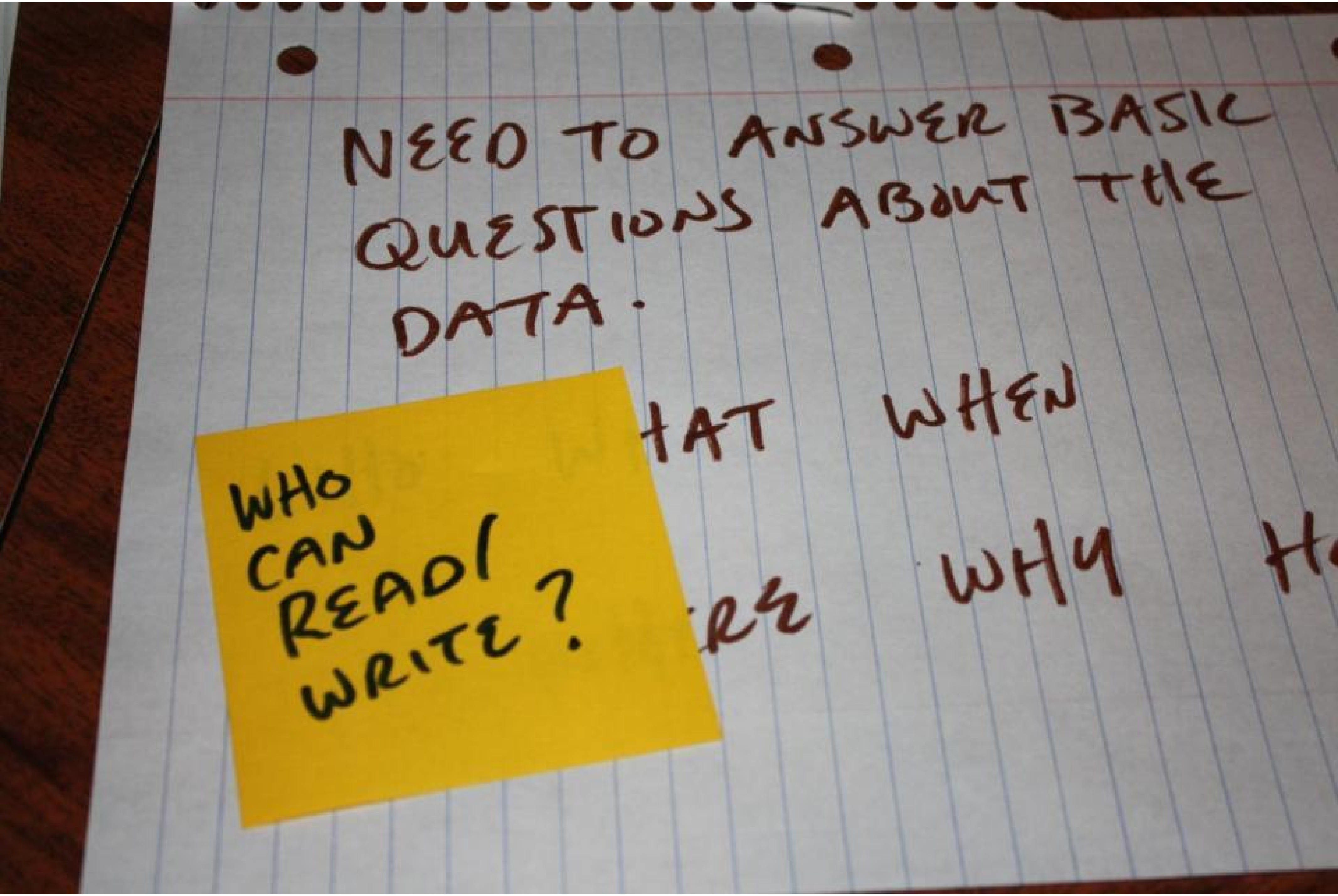

SEQUENTIAL
=
NOT
CONCURRENT

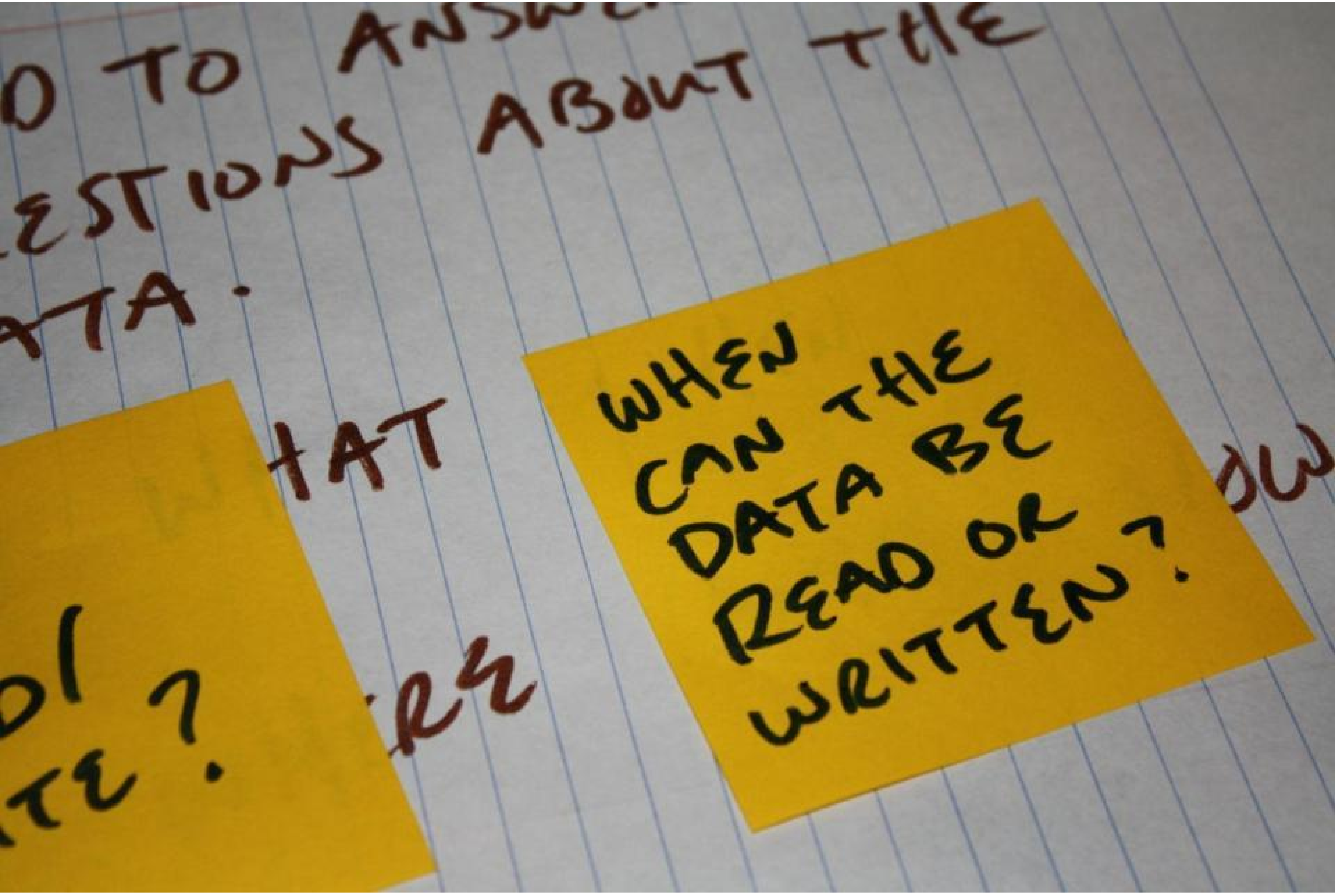
But how do you know what the conflicts are?

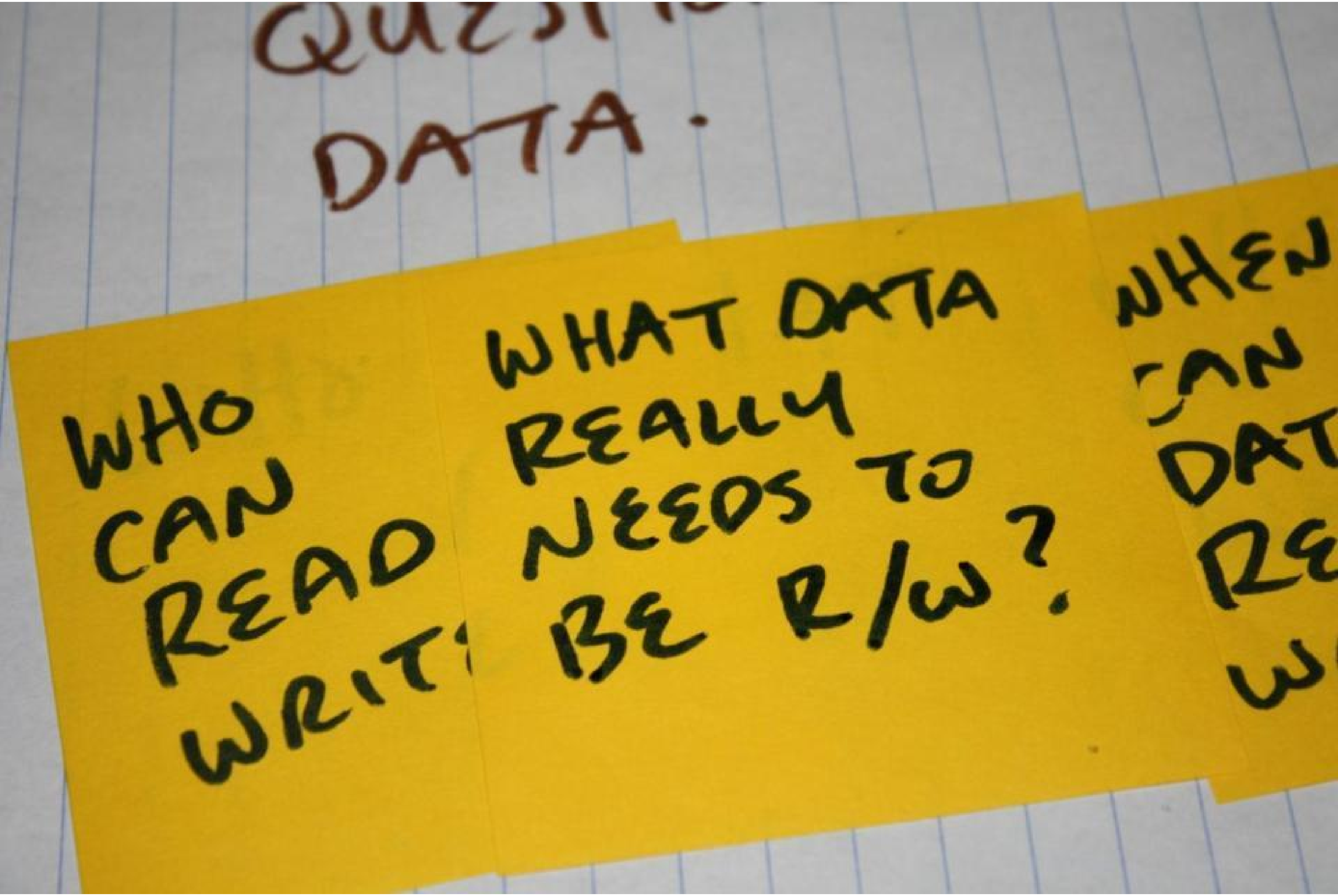
NEED TO ANSWER BASIC QUESTIONS ABOUT THE DATA.

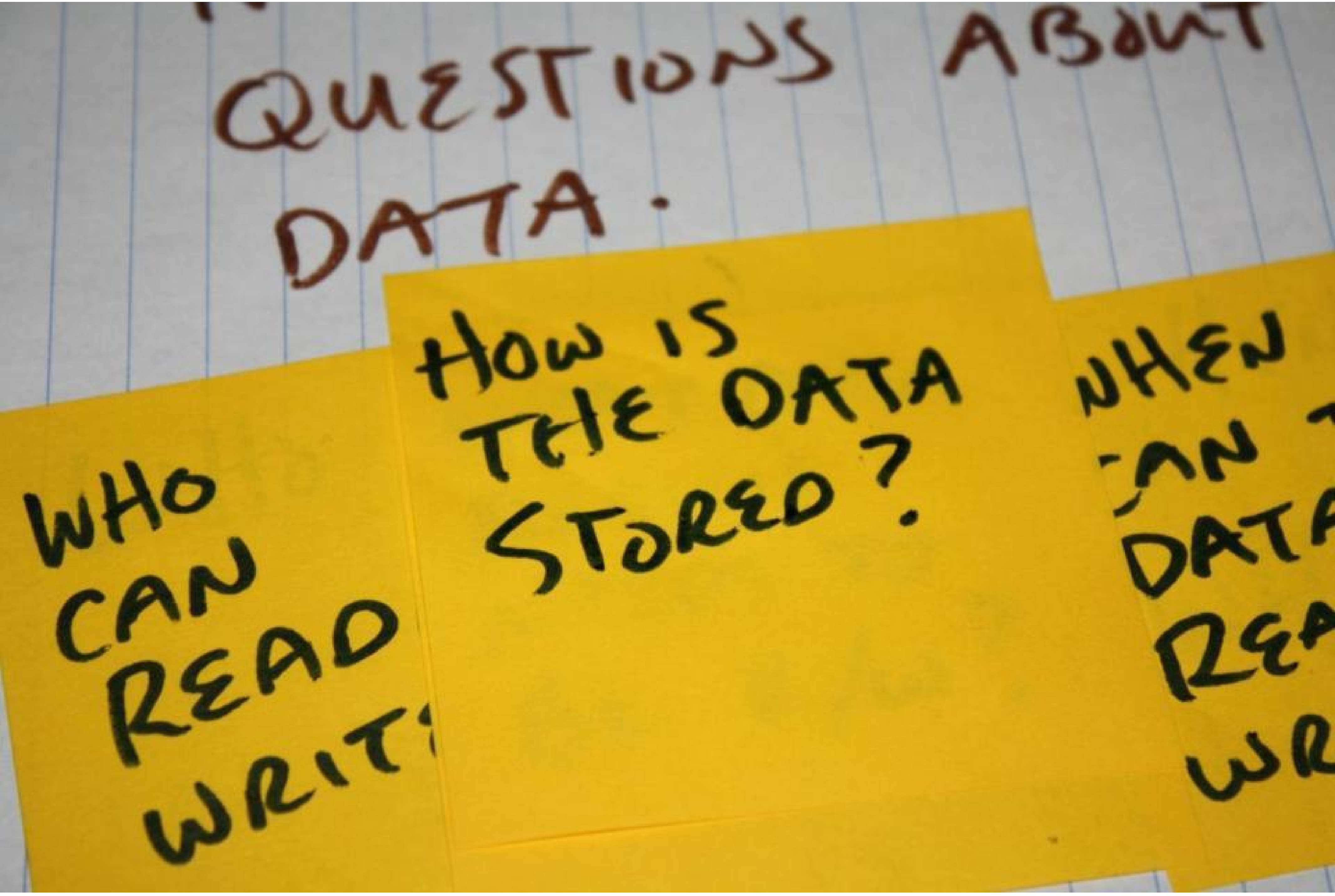
WHO WHAT WHEN

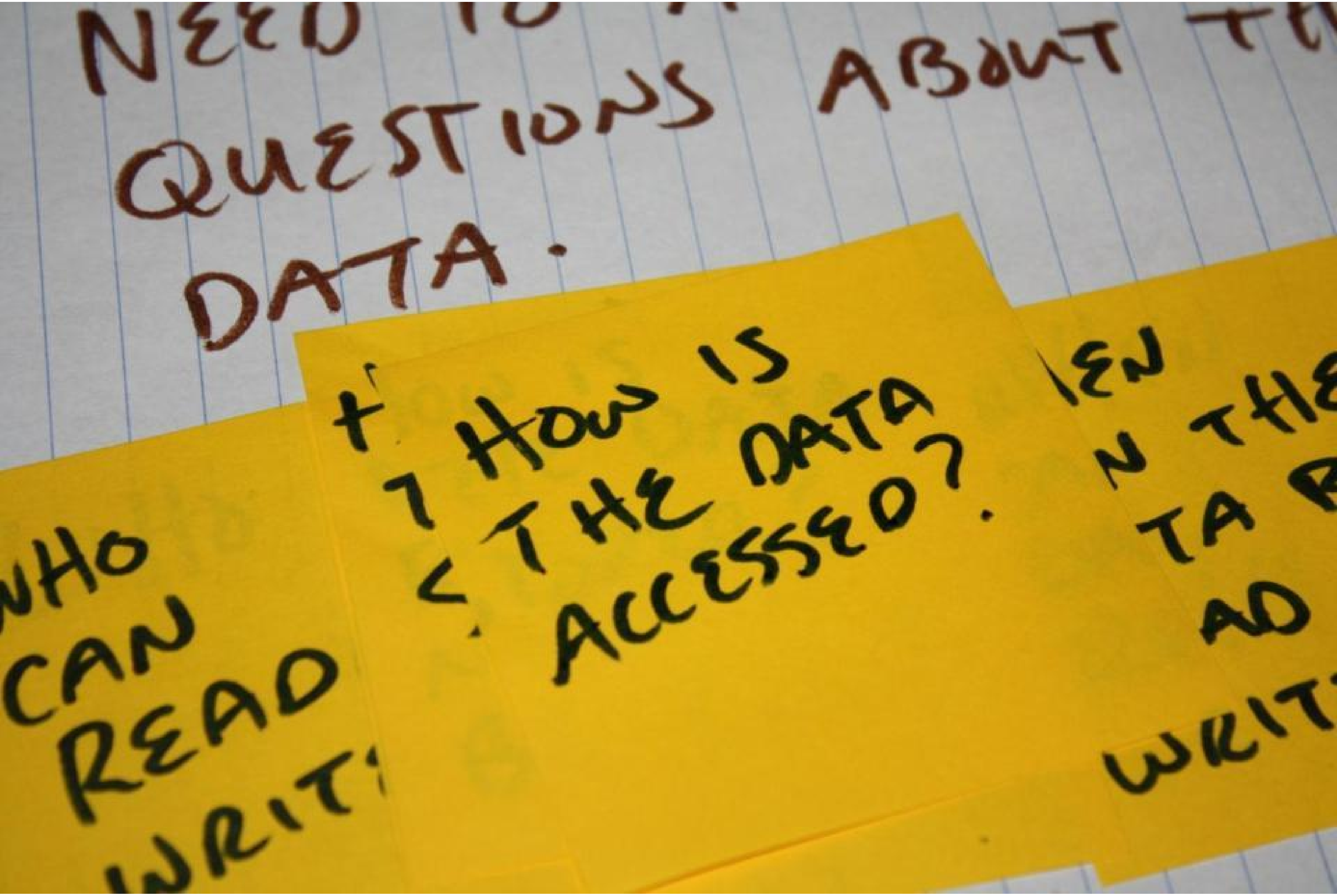
WHERE WHY HOW

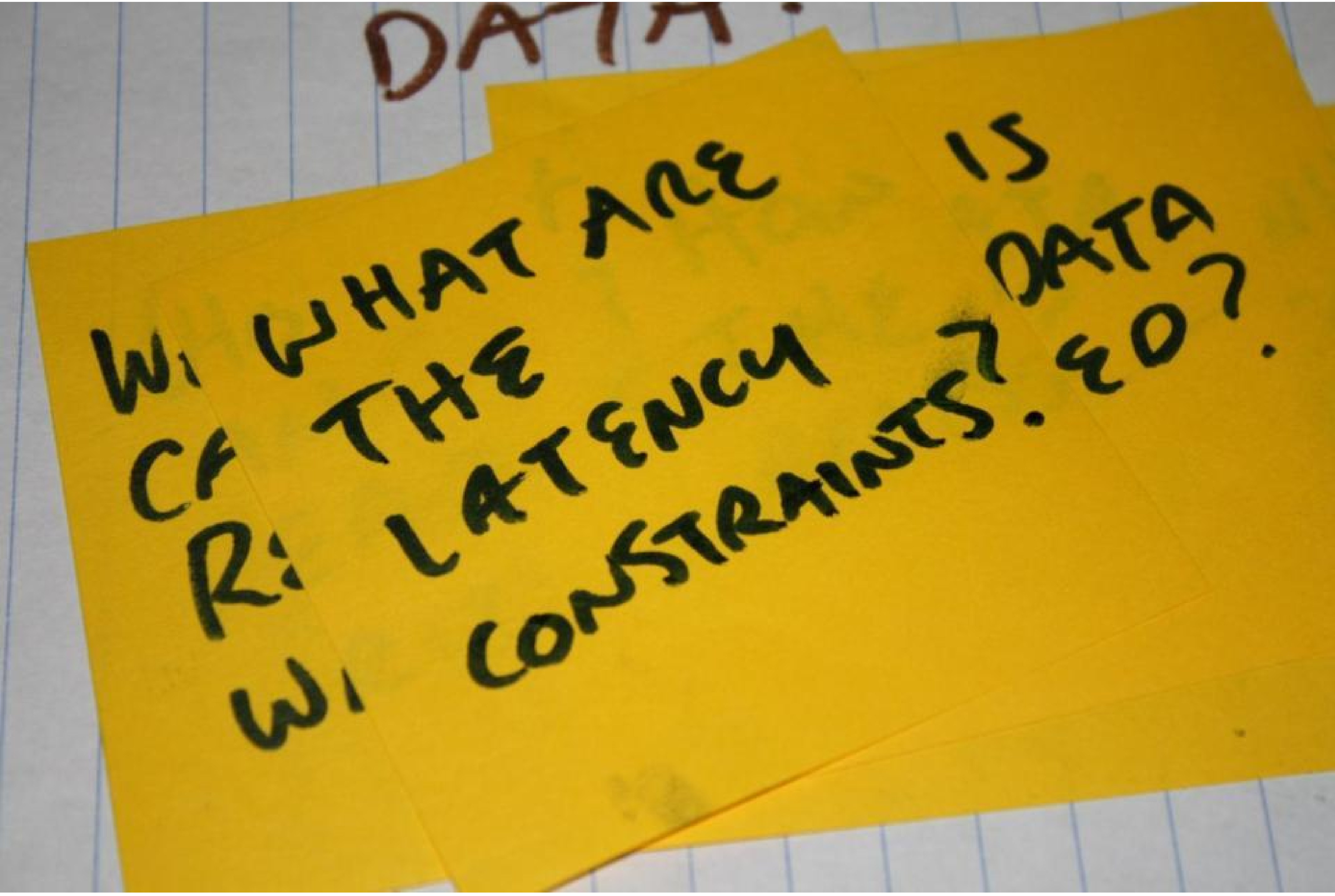

NEED TO ANSWER BASIC
QUESTIONS ABOUT THE
DATA.
WHEN
WHY
WHO
CAN
READ/
WRITE?

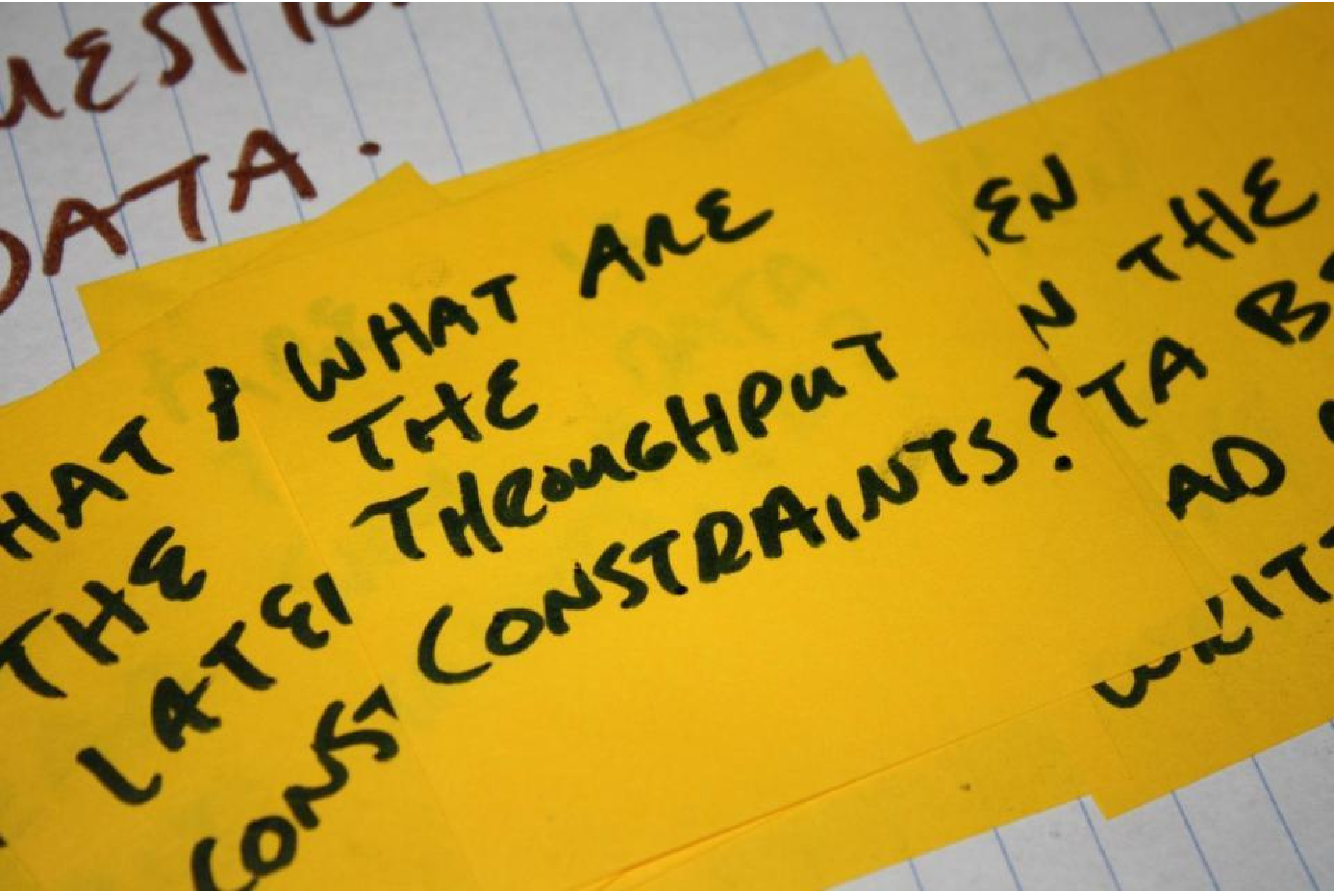

WHEN CAN THE DATA BE READ OR WRITTEN?

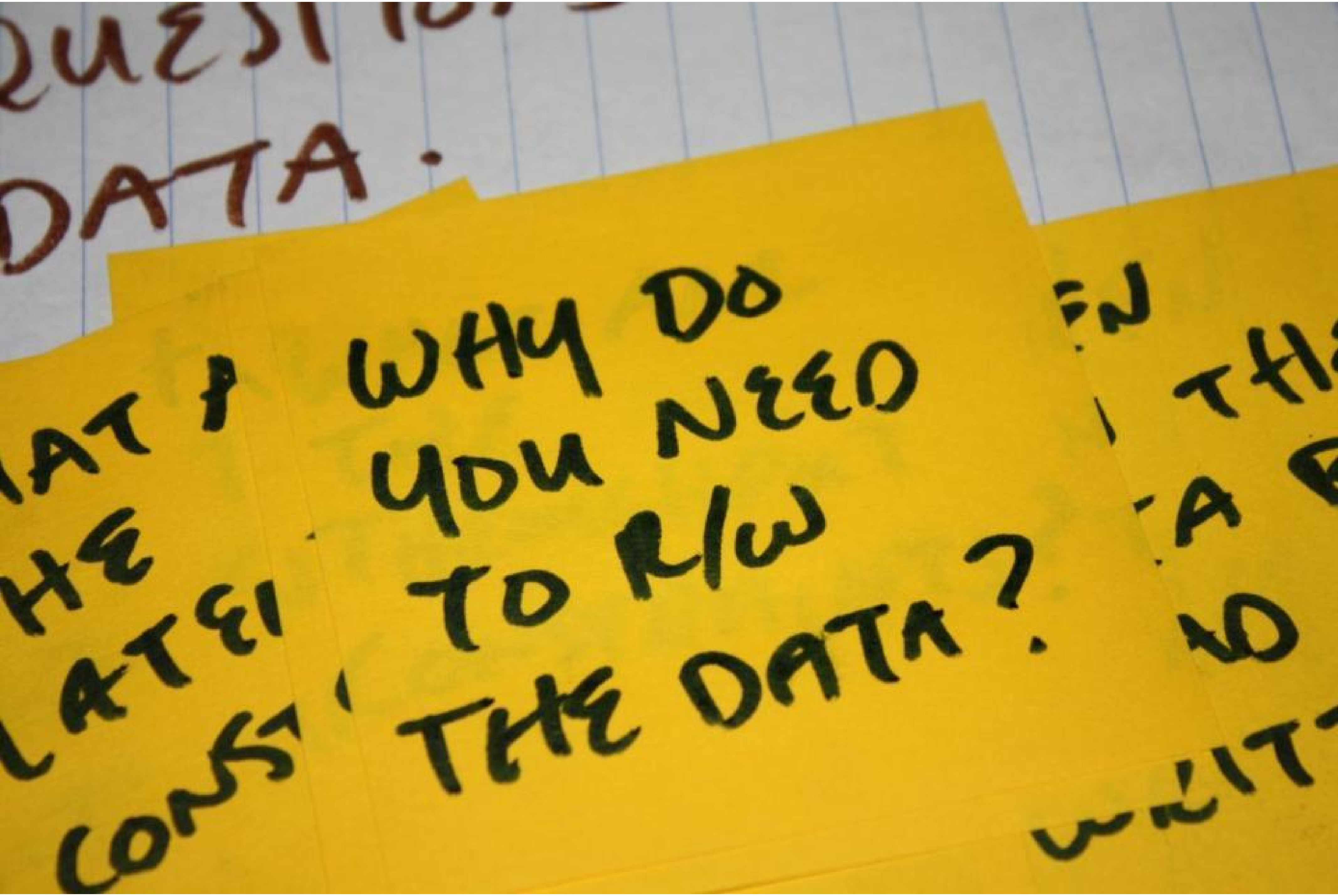

DATA:
WHO CAN READ
WHAT DATA REALLY NEEDS TO BE R/W?
WHEN CAN


QUESTIONS ABOUT
DATA.
HOW IS
THE DATA
STORED?
WHO
CAN
READ


NEED
QUESTIONS ABOUT
DATA.
HOW IS THE DATA ACCESSED?
WHO CAN READ


WHAT ARE THE LATENCY CONSTRAINTS?
IS DATA


DATA.
WHAT ARE THE THROUGHPUT CONSTRAINTS?


DATA.
WHY DO YOU NEED TO R/W THE DATA?

i.e. Understand the data!

(It always comes down to this)

Defining an concurrent insert operation:

What would it mean?

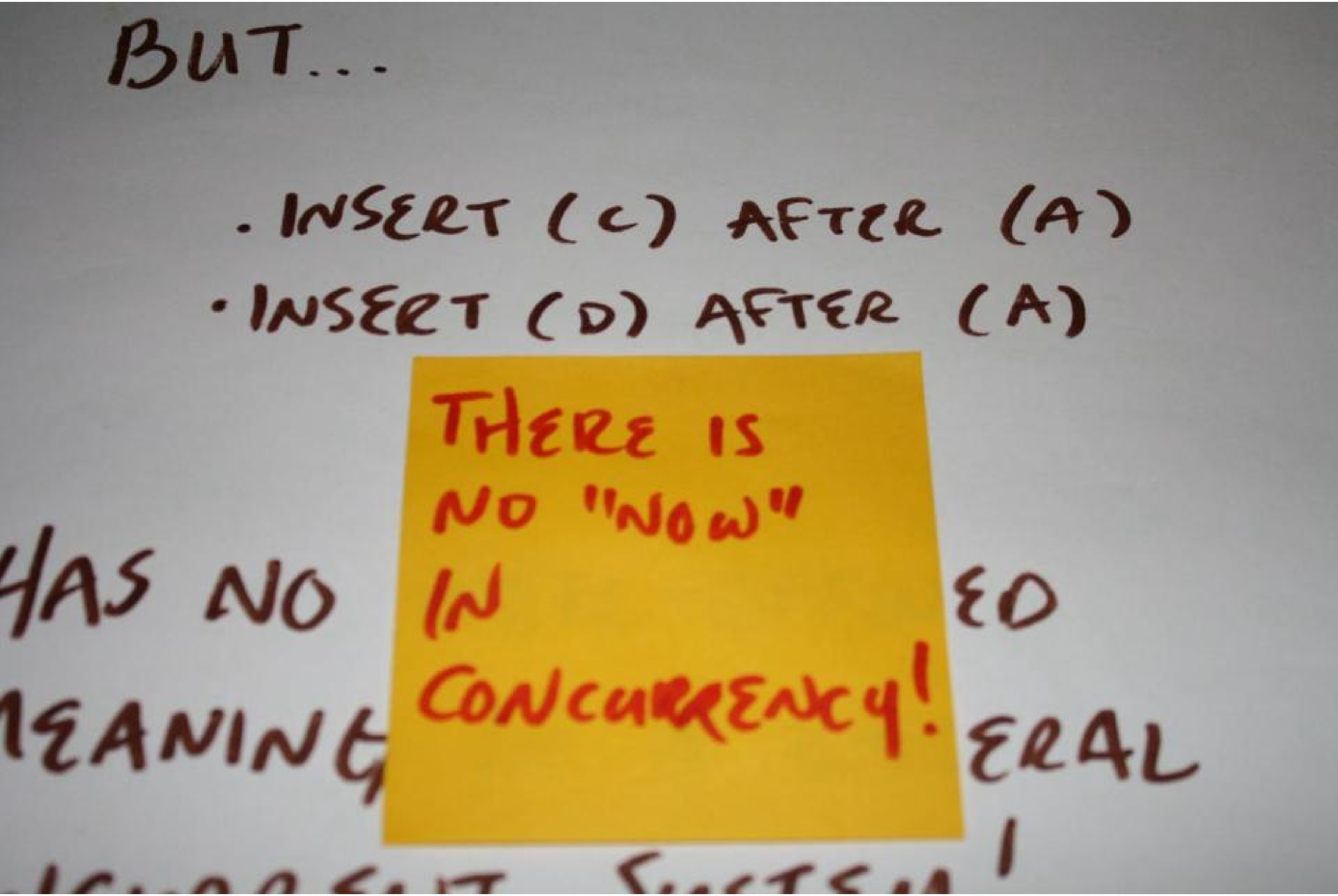
CONCURRENT INSERT OP:

WHAT DOES IT
MEAN IN CONTEXT?

BUT...

- INSERT (C) AFTER (A)
- INSERT (D) AFTER (A)

HAS NO WELL-DEFINED MEANING IN A GENERAL CONCURRENT SYSTEM!


BUT...
· INSERT (C) AFTER (A)
· INSERT (D) AFTER (A)
THERE IS NO "NOW" IN CONCURRENCY!

BUT...

- INSERT (C) AFTER (A)
- INSERT (D) AFTER (A)

W/OUT "NOW", THERE'S NO BEFORE/AFTER

HAS NO ... ED

MEANING ... ERAL

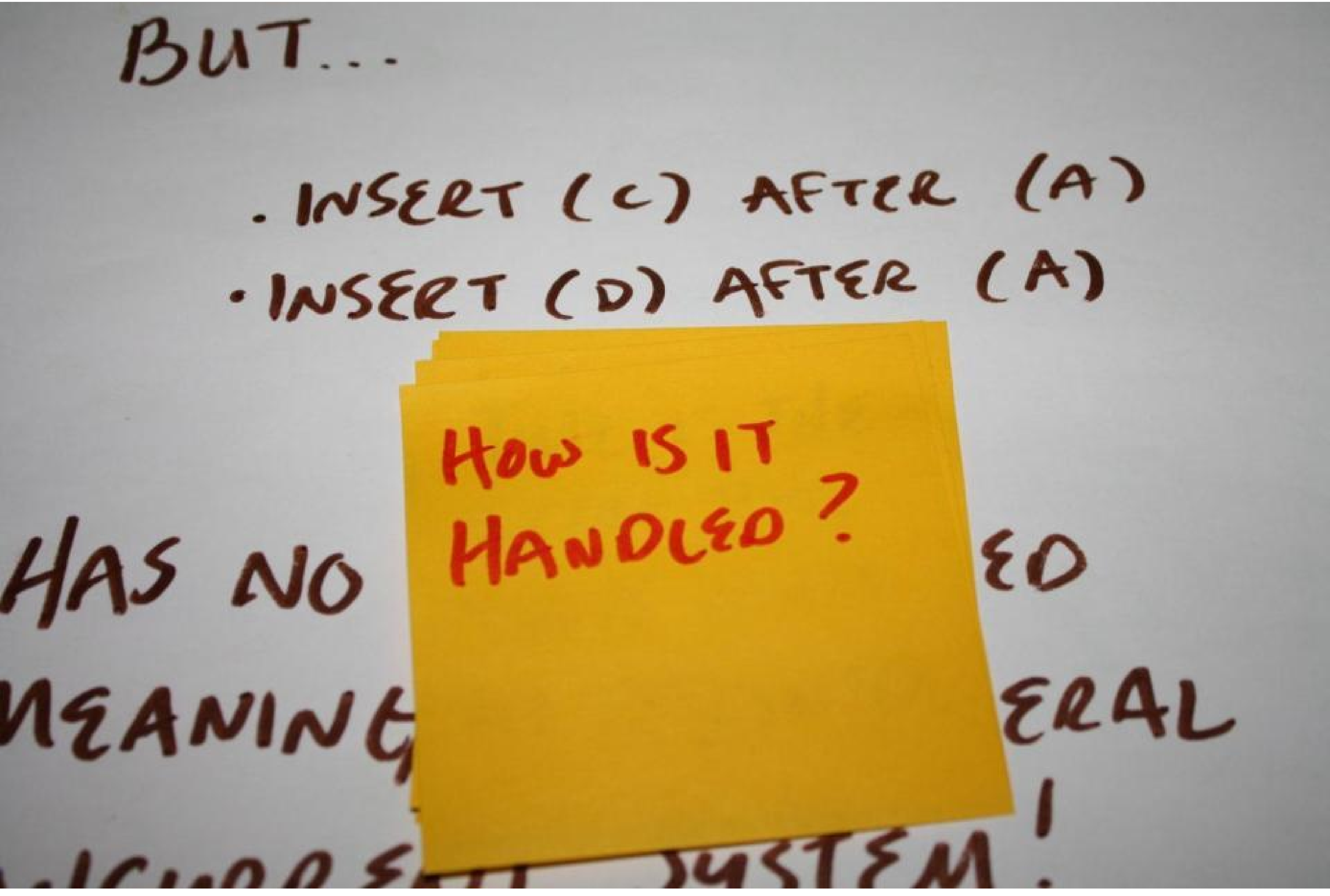
...NCURRENT SYSTEM!

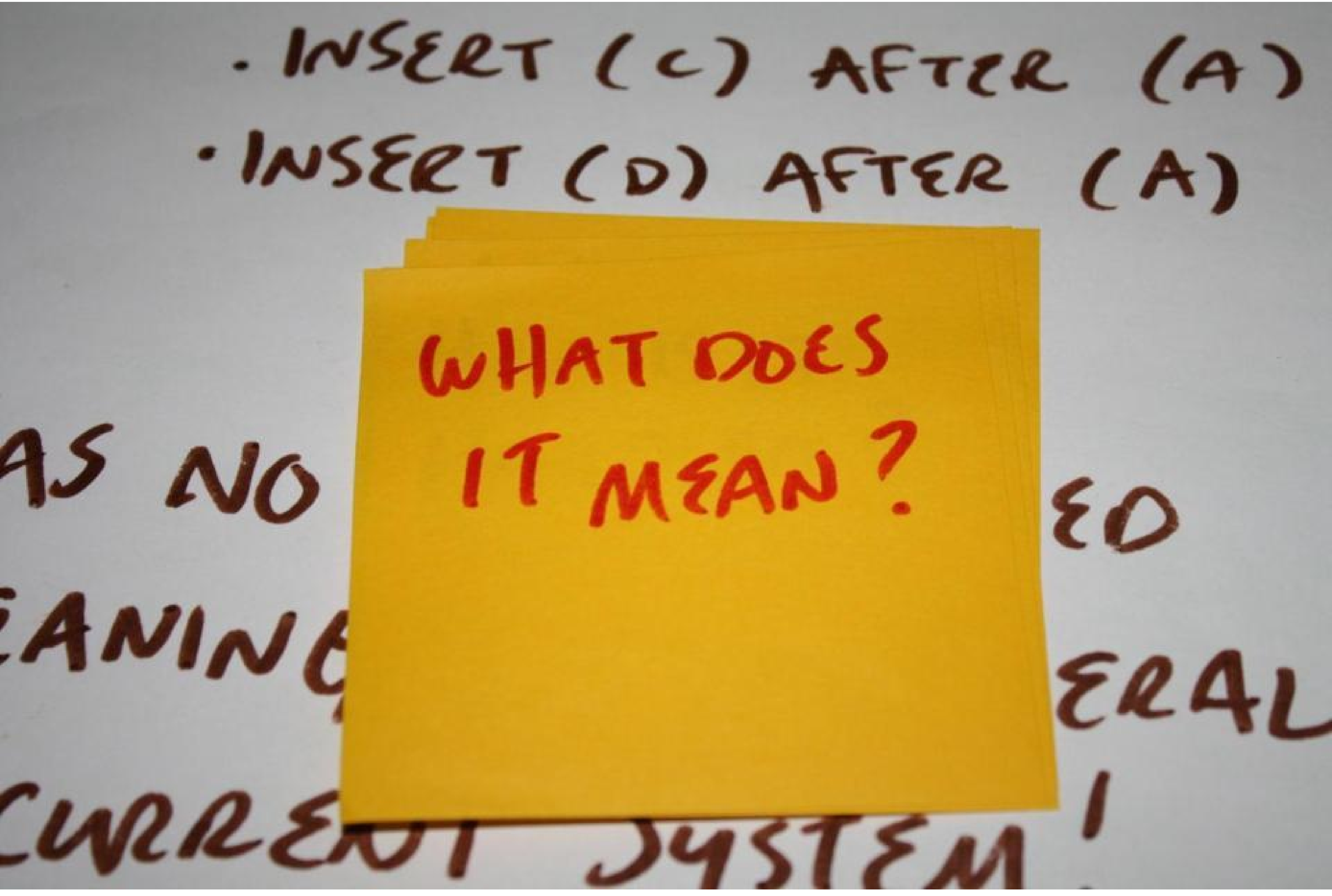
BUT...

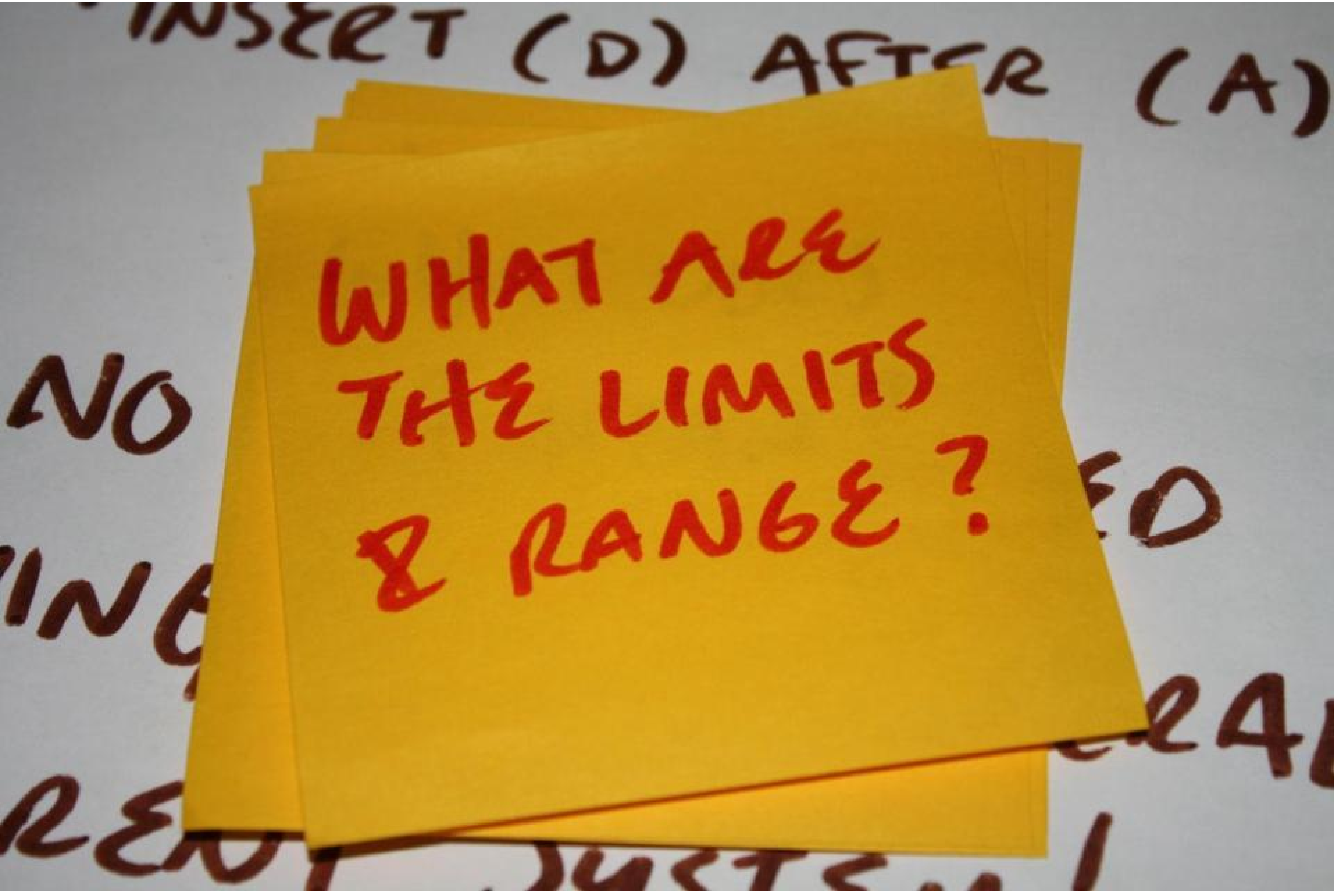
· INSERT (C) AFTER (A)
· INSERT (D) AFTER (A)

TIME IS THE
EXTRA DIM
THAT MUST
BE DEFINED.

HAS NO ... ED
EANING ... ERAL
NCURRENT SYSTEM!

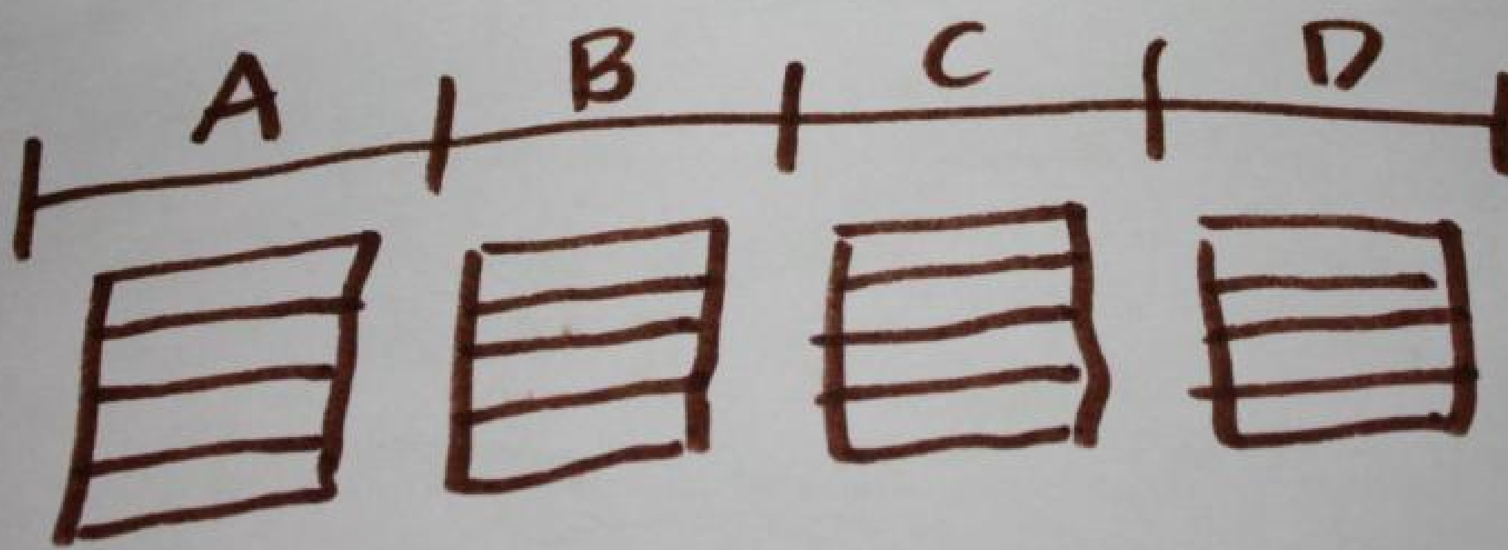

BUT...
· INSERT (C) AFTER (A)
· INSERT (D) AFTER (A)
HOW IS IT HANDLED?
HAS NO


· INSERT (C) AFTER (A)
· INSERT (D) AFTER (A)
WHAT DOES IT MEAN?

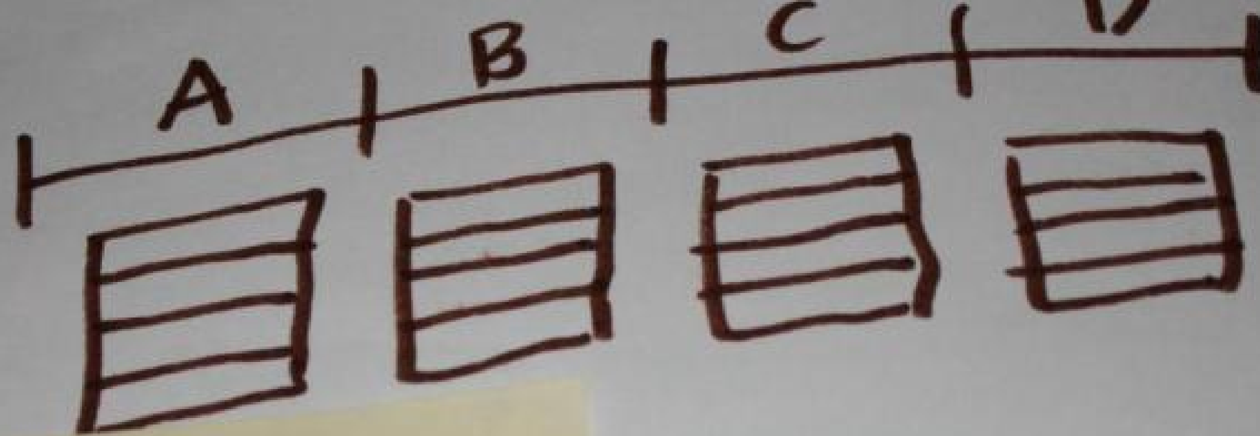

WHAT ARE
THE LIMITS
& RANGE?

How might you answer these questions?

WHAT IS THE ACCURACY / GRANULARITY OF GLOBAL ORDER VALUES?

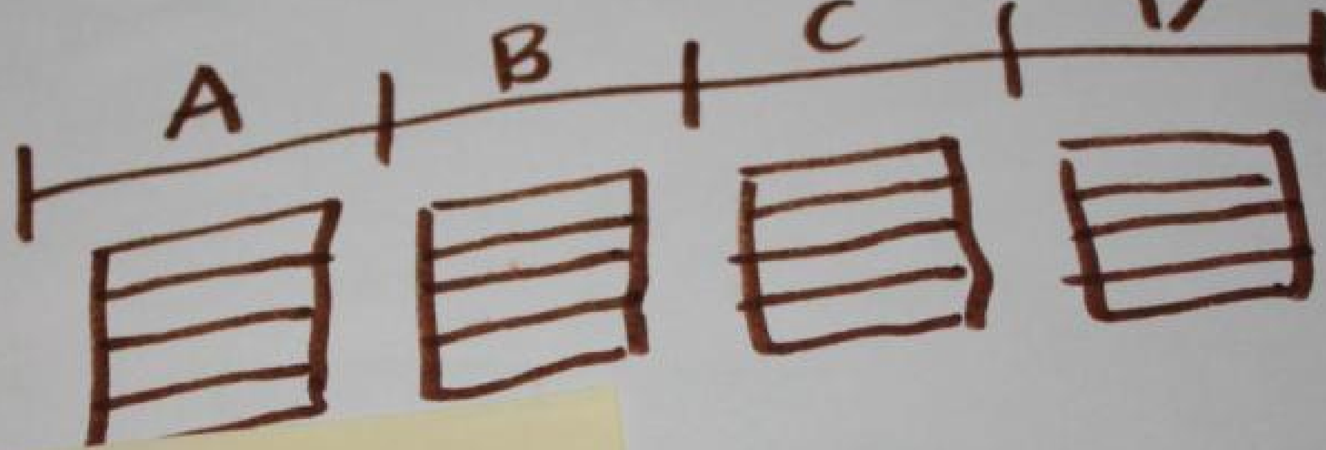


GRANULARITY O
ORDER VALUES?

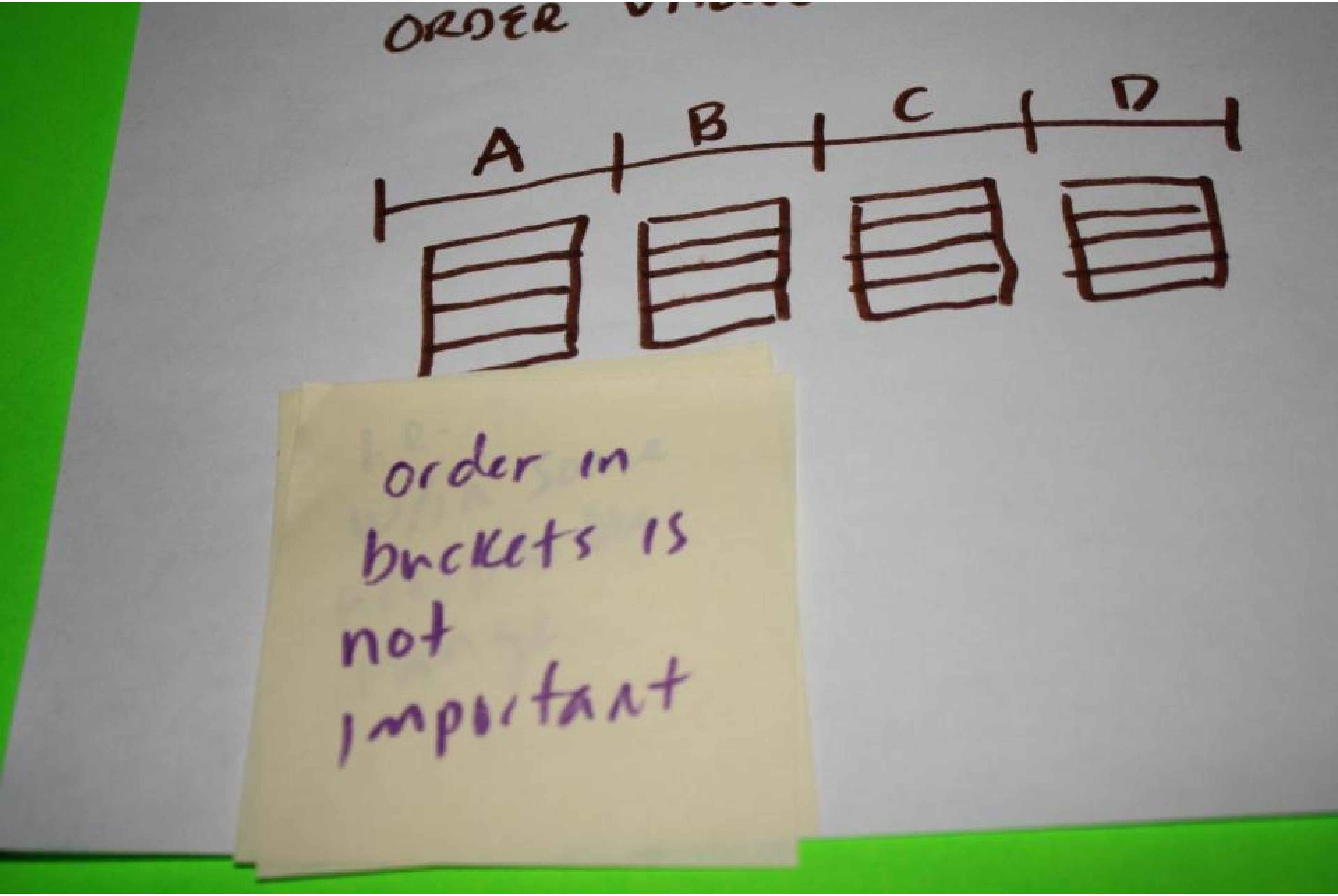


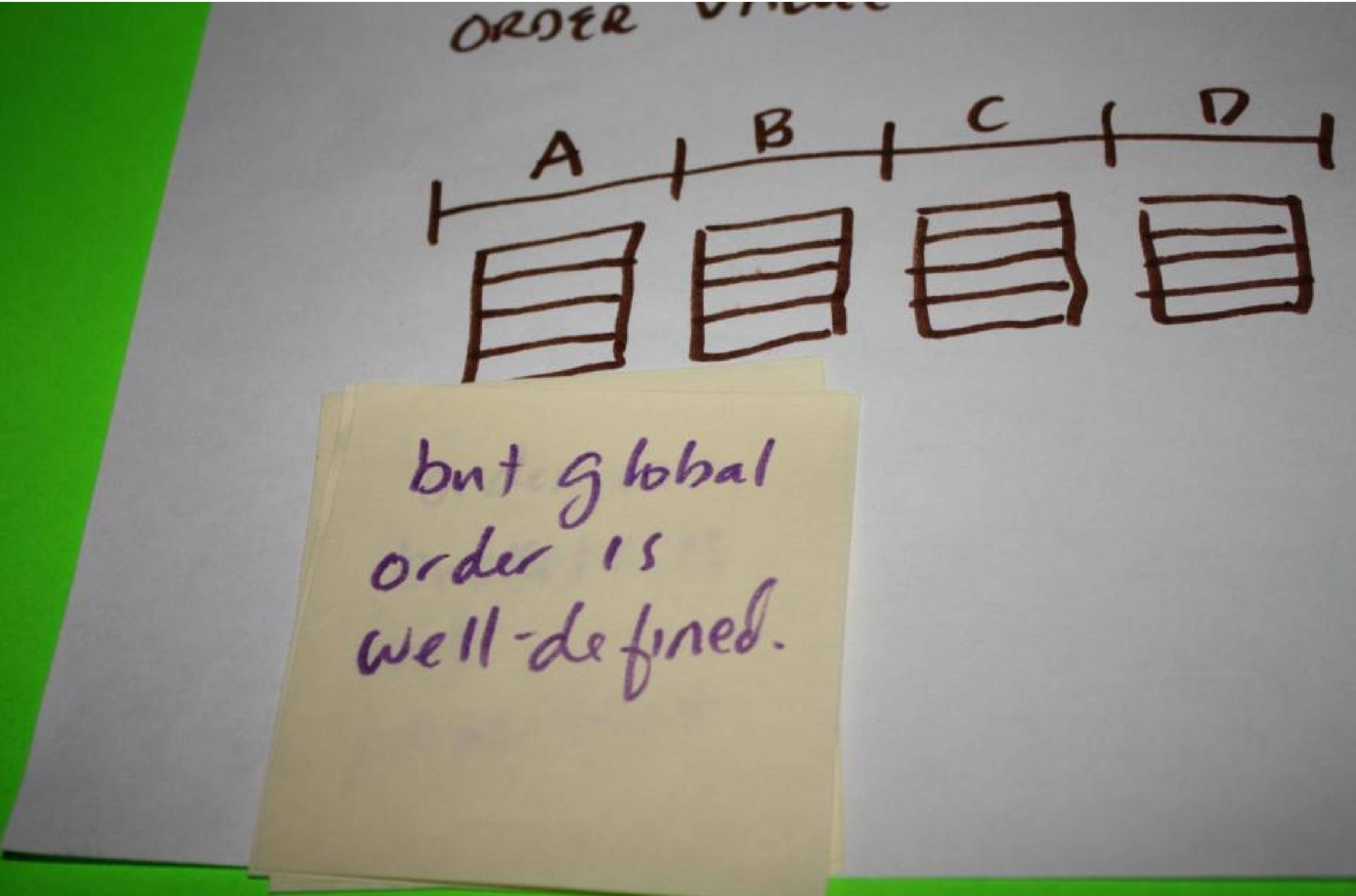
Maybe
"Insert A"
means in
(A) bucket.

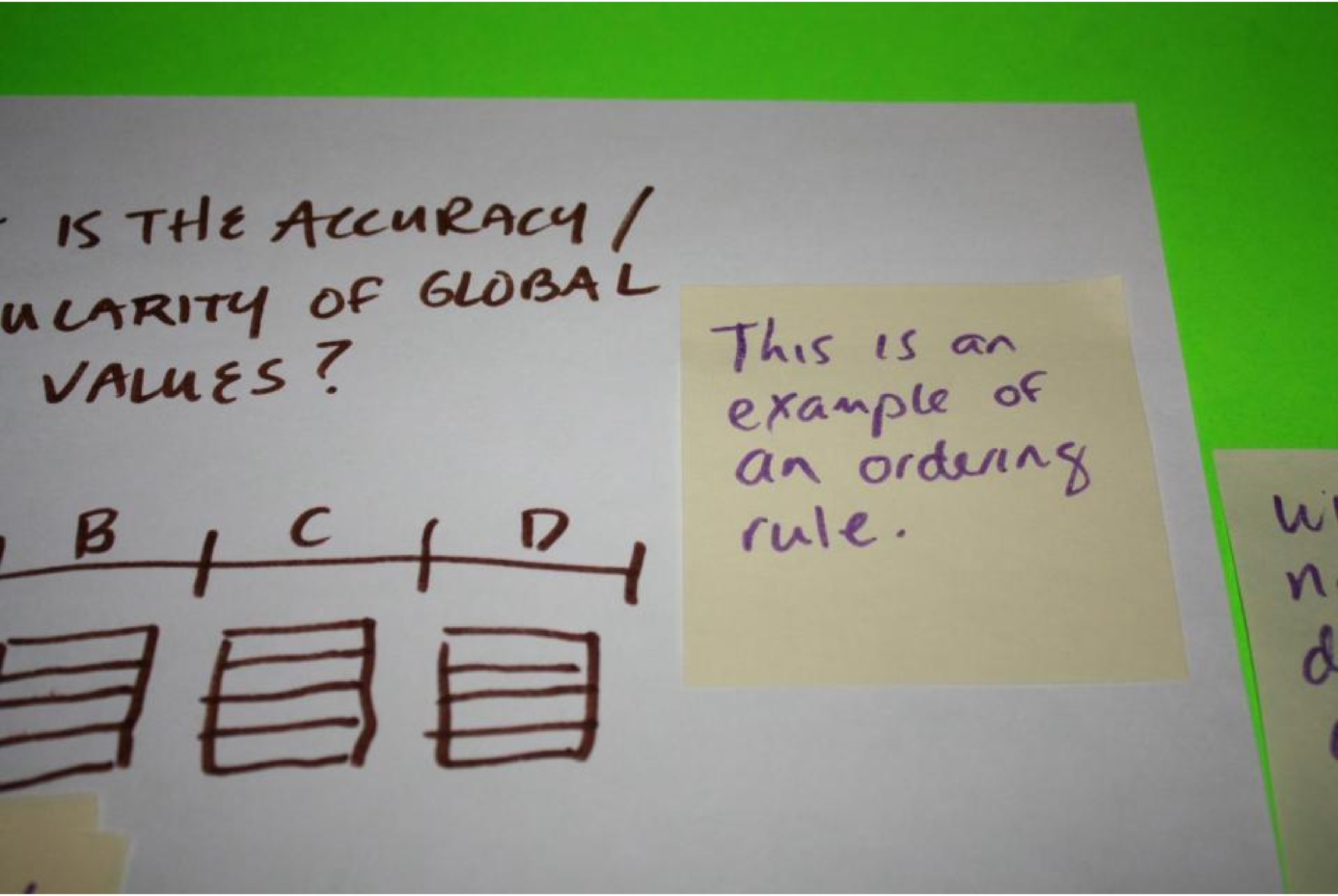
ORDER VALUES:

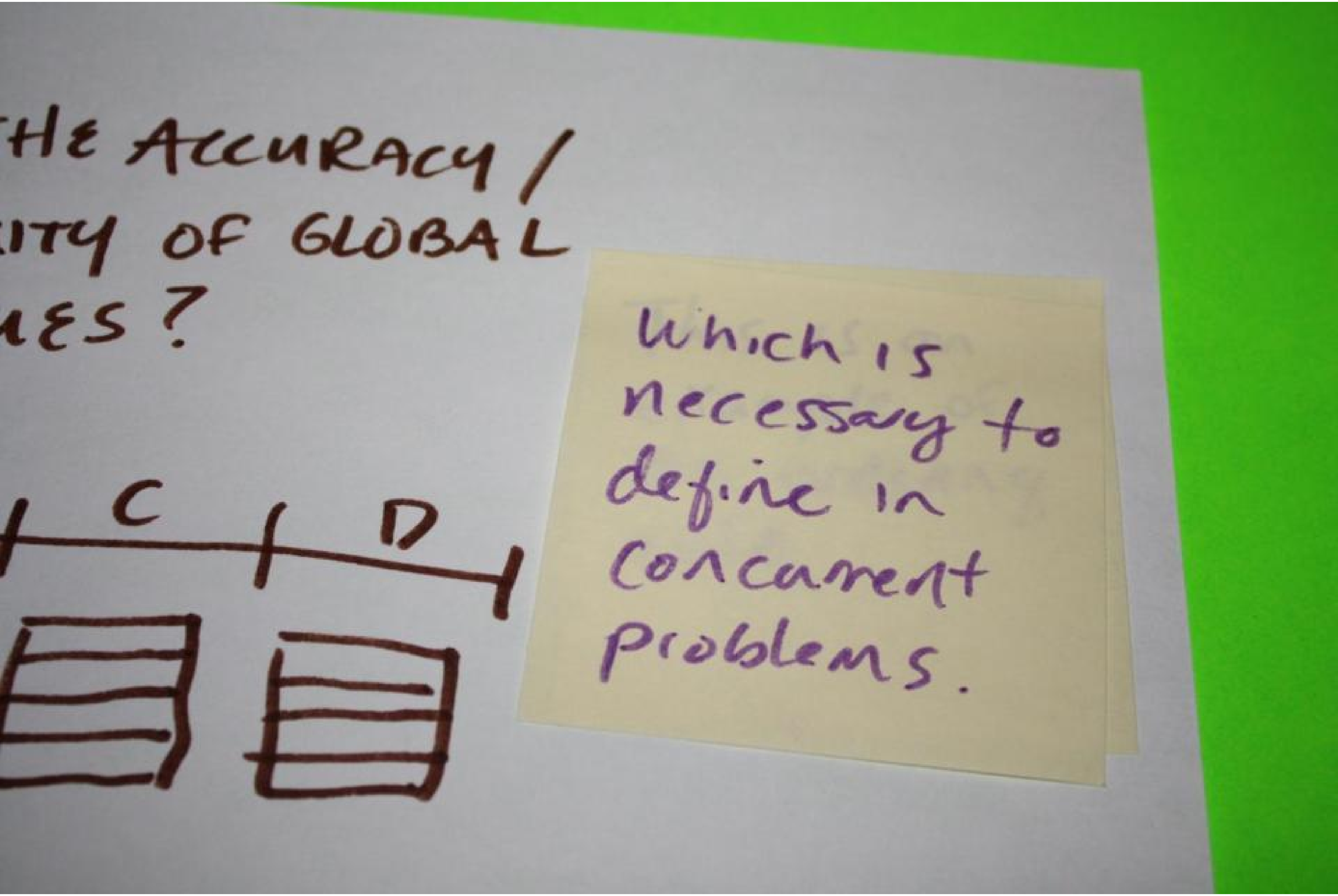


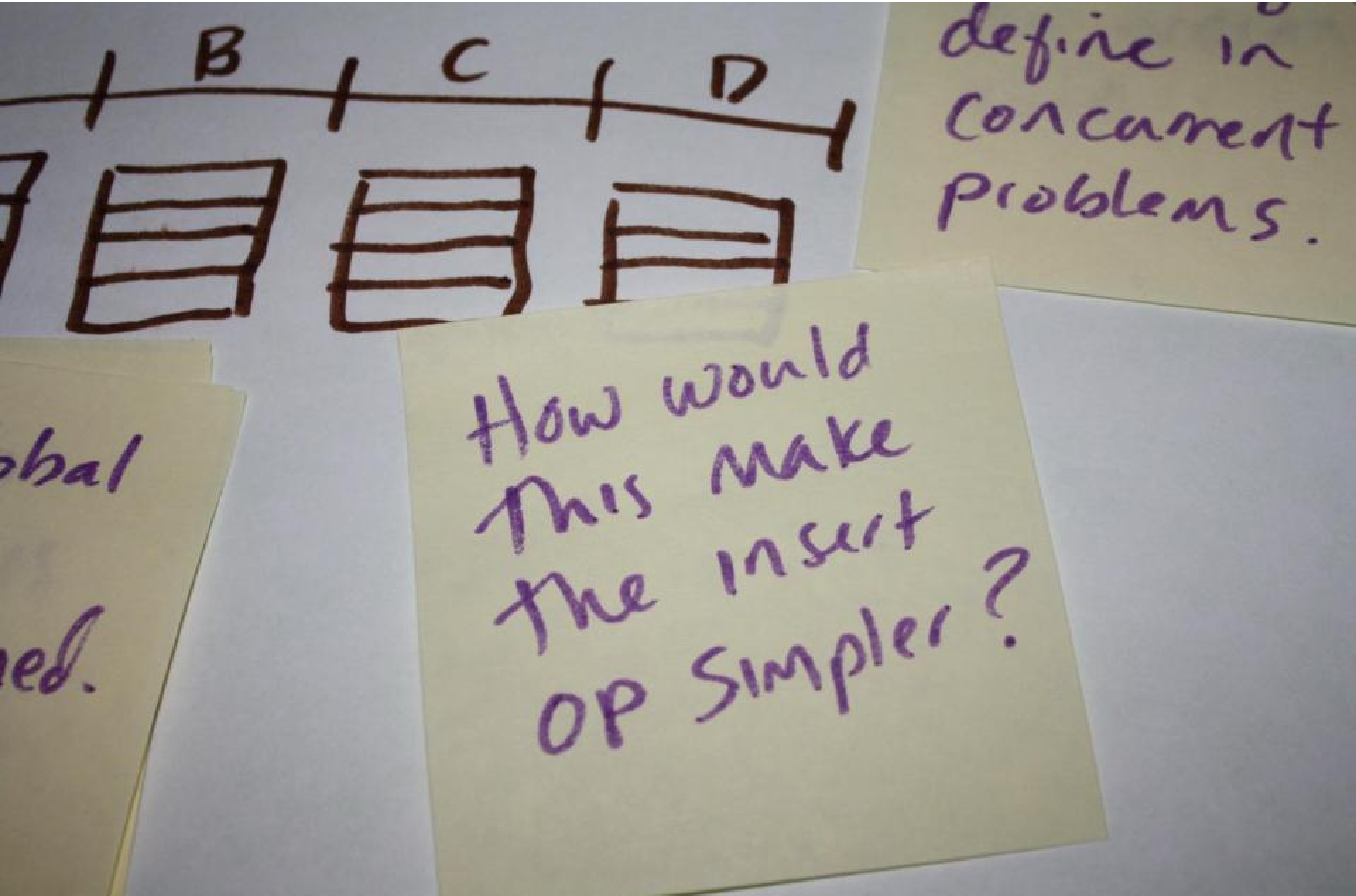
i.e.
w/in some
acceptable
range.


ORDER
A
B
C
D
order in buckets is not important

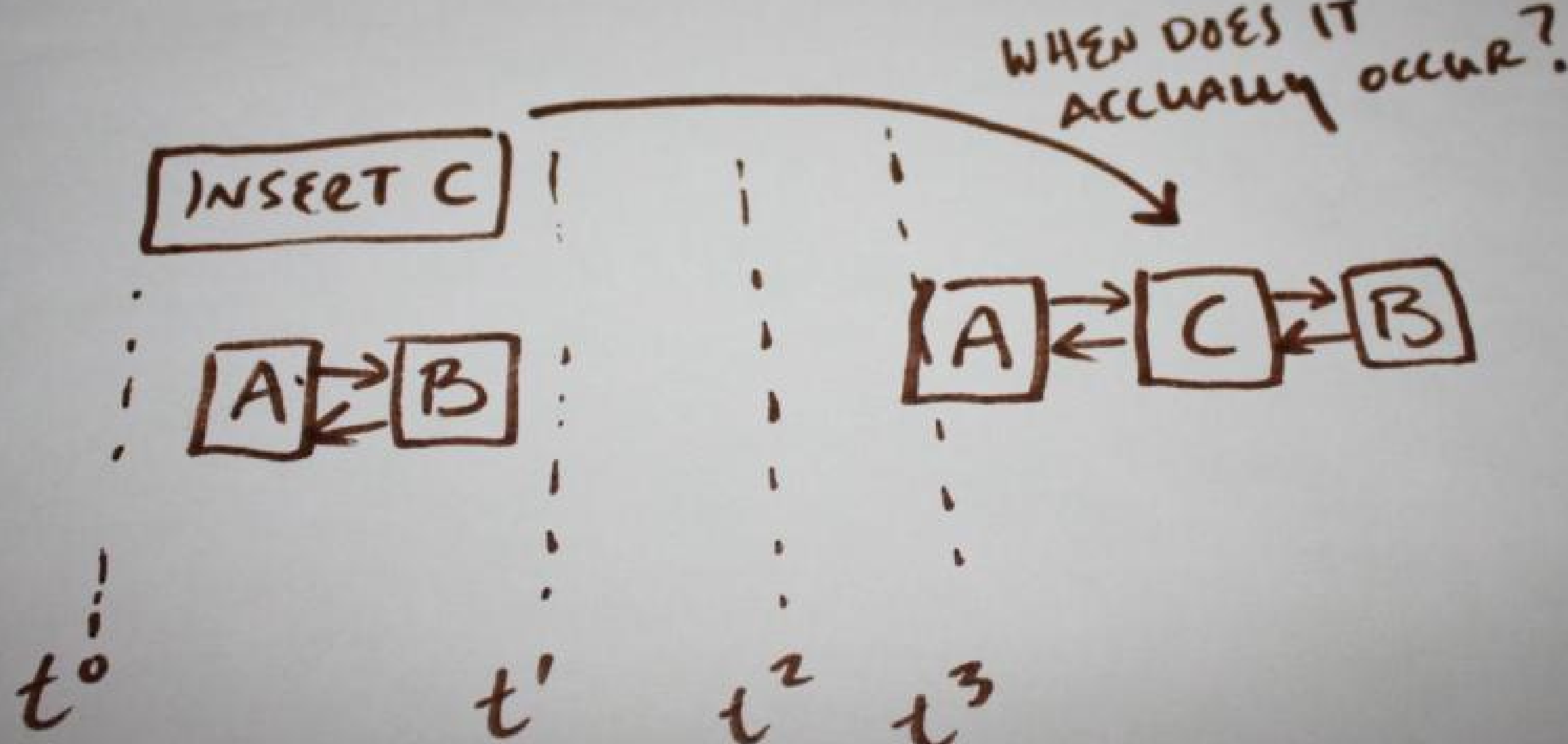

ORDER
A
B
C
D
but global order is well-defined.


IS THE ACCURACY /
ULARITY OF GLOBAL
VALUES?
B
C
D
This is an example of an ordering rule.


HE ACCURACY /
ITY OF GLOBAL
ES?
C
D
Which is necessary to define in concurrent problems.


B
C
D
define in
concurrent
problems.
obal
ned.
How would
this make
the insert
op simpler?

WHAT ARE THE LATENCY REQUIREMENTS OF THE INSERT OP?



LESSON

DOUBLY-LINKED LIST IS
SEQUENTIAL DATA STRUCT.

PRESUMES:

- ZERO LATENCY
- GUARANTEED ORDER
  (SEQUENTIAL)

CONCURRENT DATA STRUCTURES DEFINED BY:

- EXPLICIT LATENCY REQUIREMENTS
- EXPLICIT ORDERING RULES

But these requirements and rules can only be defined **in context**

WHAT ARE SOME
CONTEXTS W/IN YOU
WOULD USE DOUBLY-
LINKED LIST?

WHAT ARE SOME CONTEXTS W/IN YOU WOULD USE DOUBLY-LINKED LIST?

I.E.
DON'T TRY
TO FIT THE
"SOLUTION"
TO THE
PROBLEM.

WHAT ARE SOME CONTEXTS W/IN YOU WOULD USE DOUBLY-LINKED LIST?

i.e.

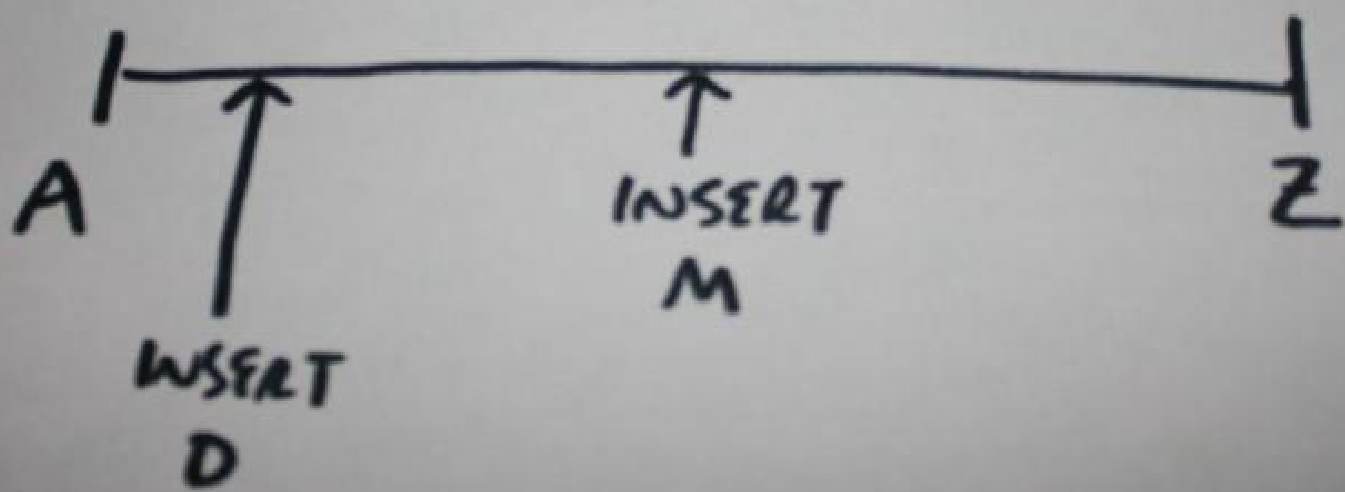
JUST SOLVE THE PROBLEM.

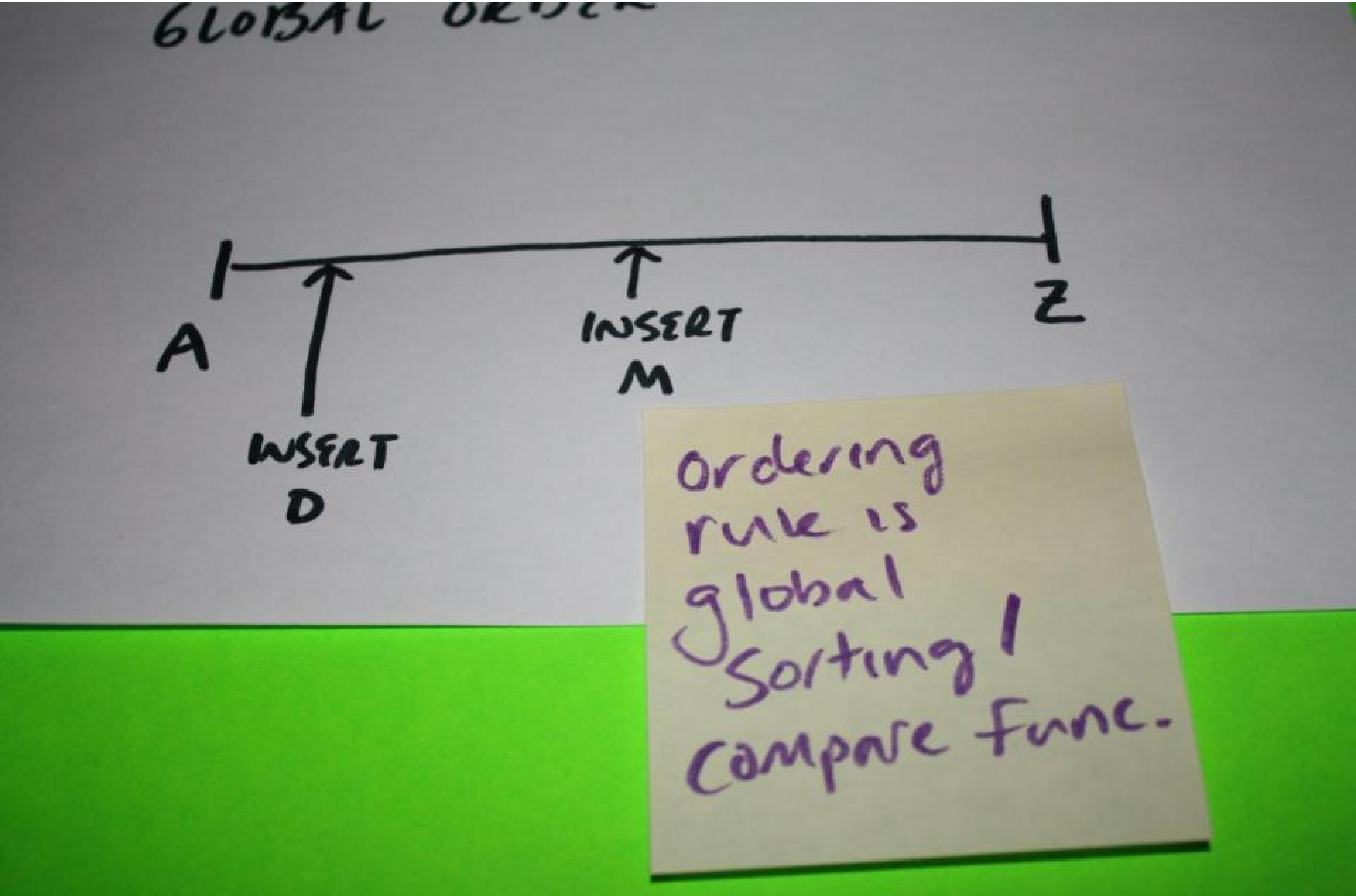
e.g. Insert
Sort

INSERT SORT MEANS

INSERT AT POSITION IN

GLOBAL ORDER




GLOBAL
A
Z
INSERT
M
INSERT
D
ordering rule is global
Sorting /
Compare func.

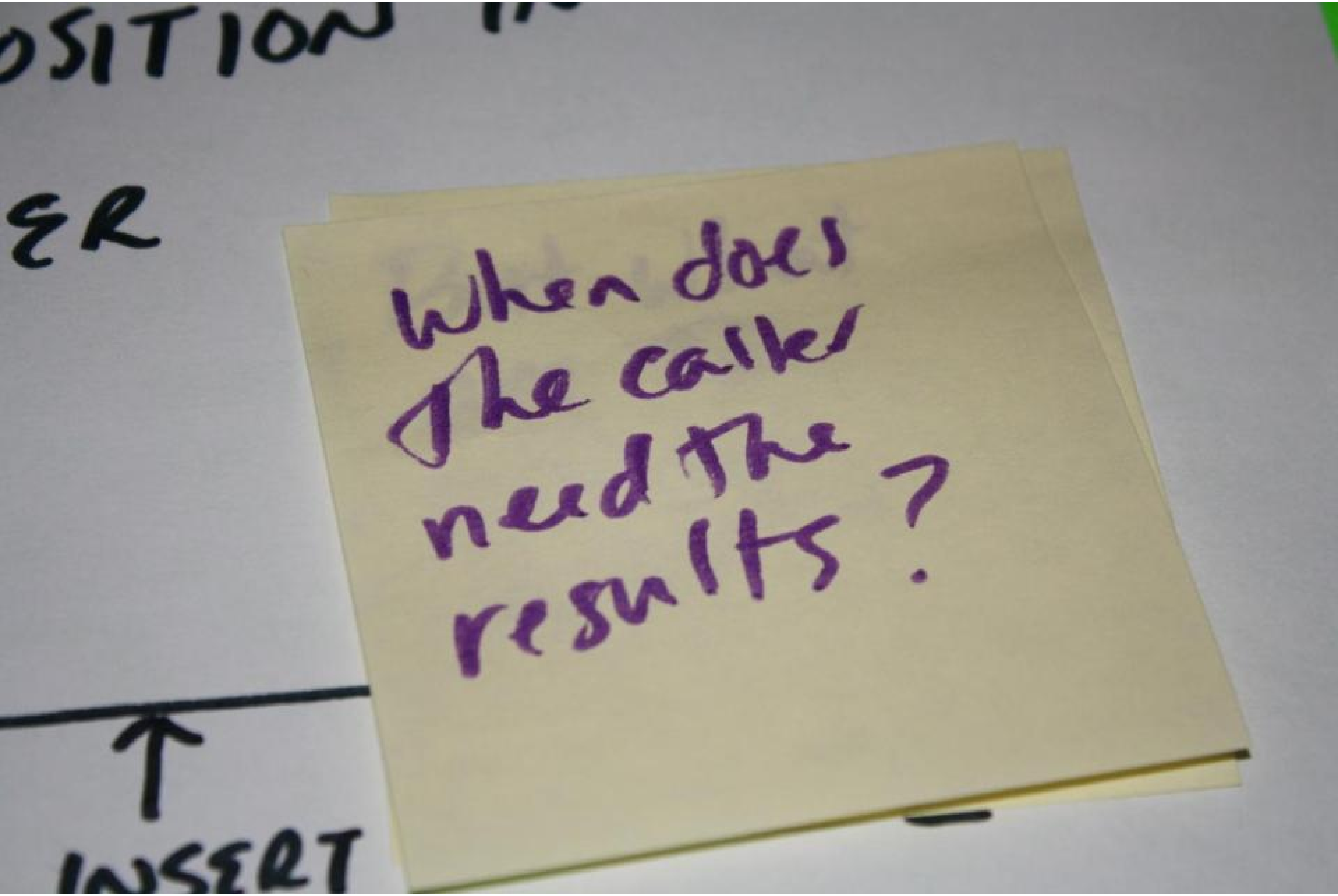
AT POSITION IN

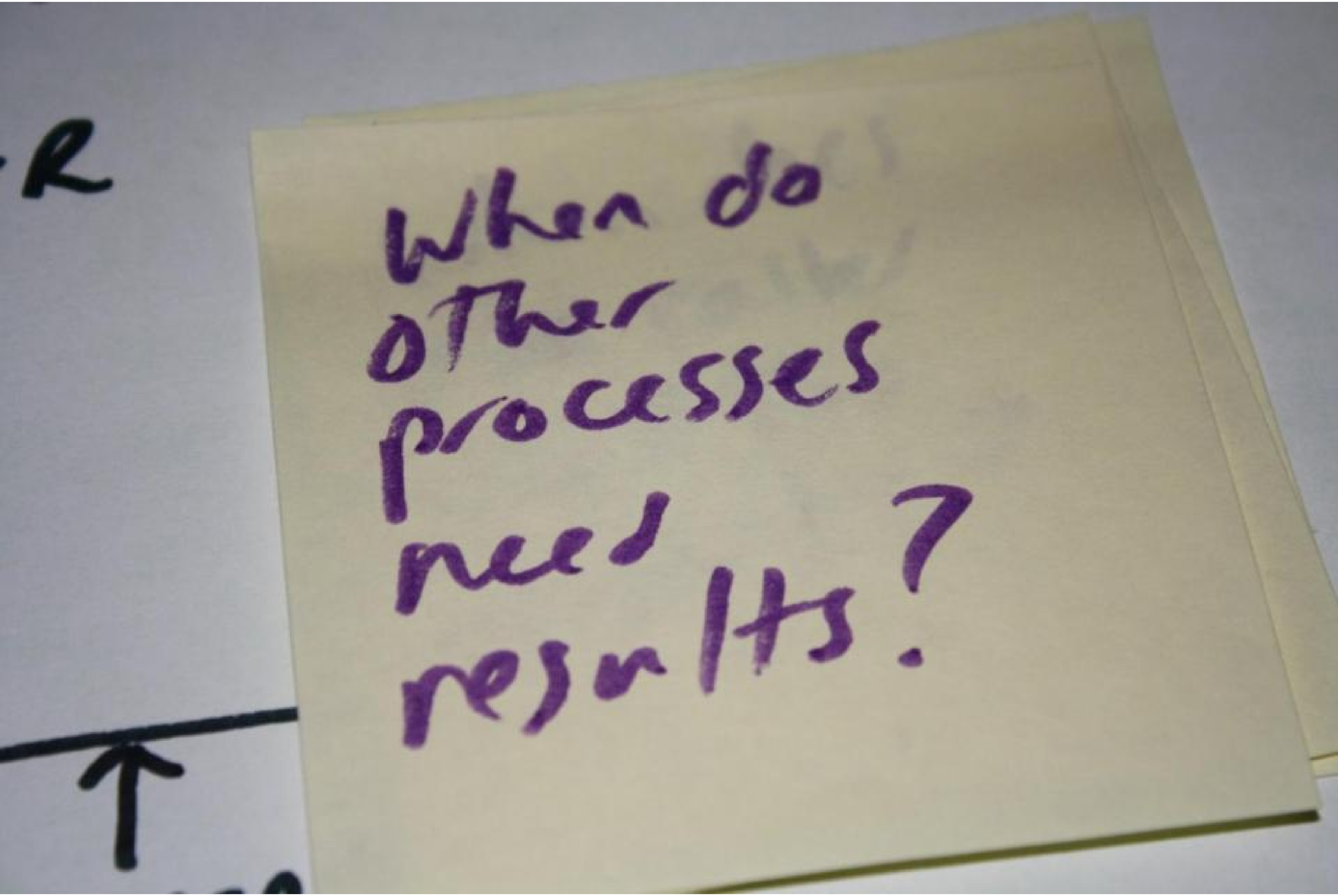
ORDER

But what are the latency rules?

INSERT
M

ordering

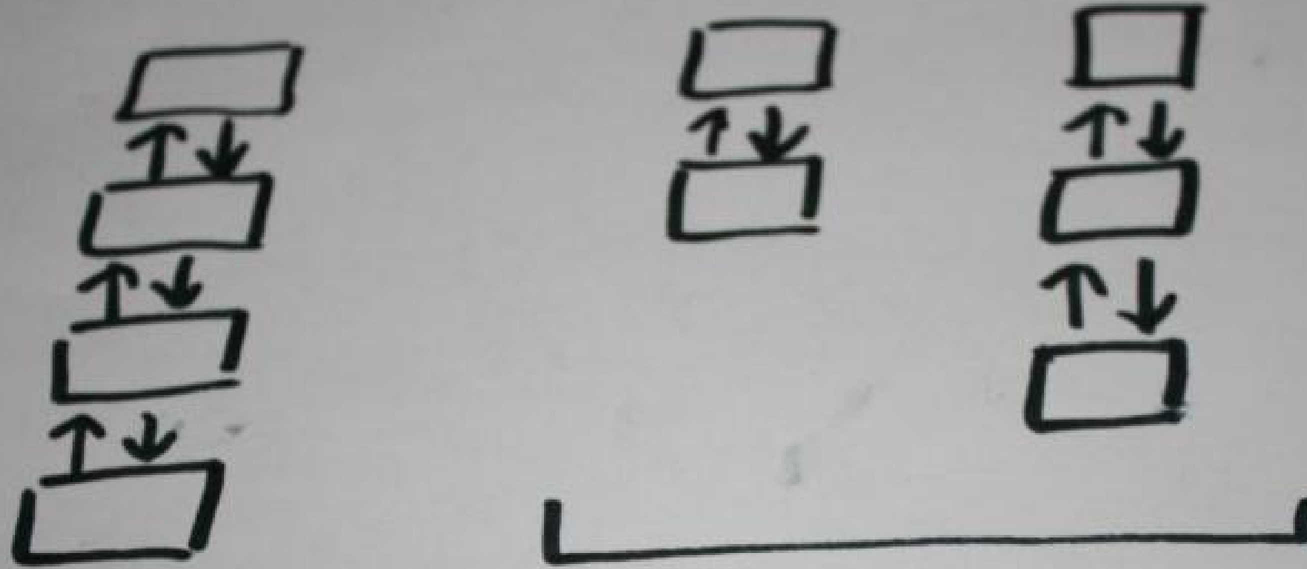

OSITION
ER
When does the caller need the results?
INSERT


When do
other
processes
need
results?


GLOBAL ORDE
Do you think dbl linked list will be right data?
INSERT

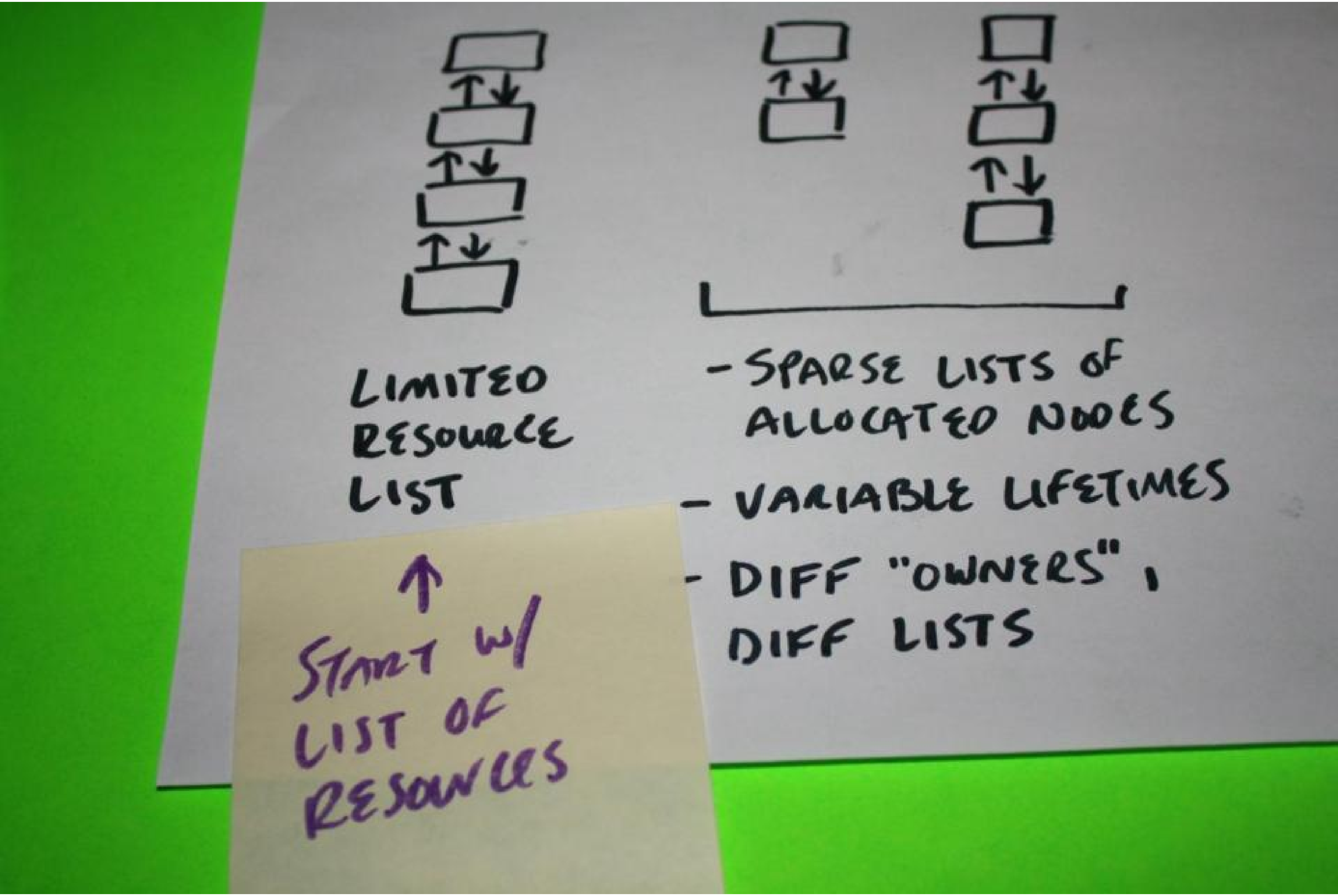
Different answers to each question would change the data structure required.

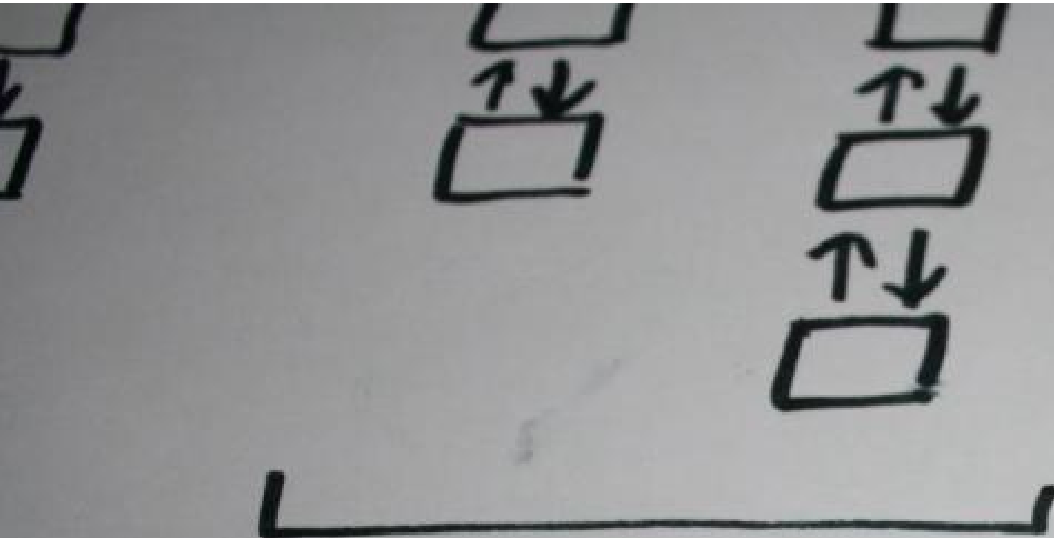
e.g. Resource Mgmt
w/ variable
life times



LIMITED
RESOURCE
LIST

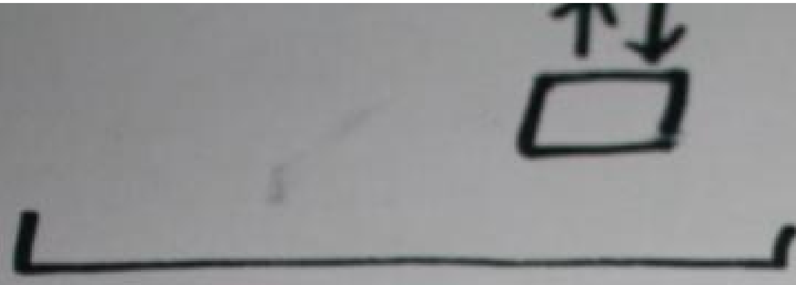
- SPARSE LISTS OF ALLOCATED NODES
- VARIABLE LIFETIMES
- DIFF "OWNERS", DIFF LISTS


LIMITED
RESOURCE
LIST
- SPARSE LISTS OF
ALLOCATED NODES
- VARIABLE LIFETIMES
- DIFF "OWNERS",
DIFF LISTS
START W/
LIST OF
RESOURCES



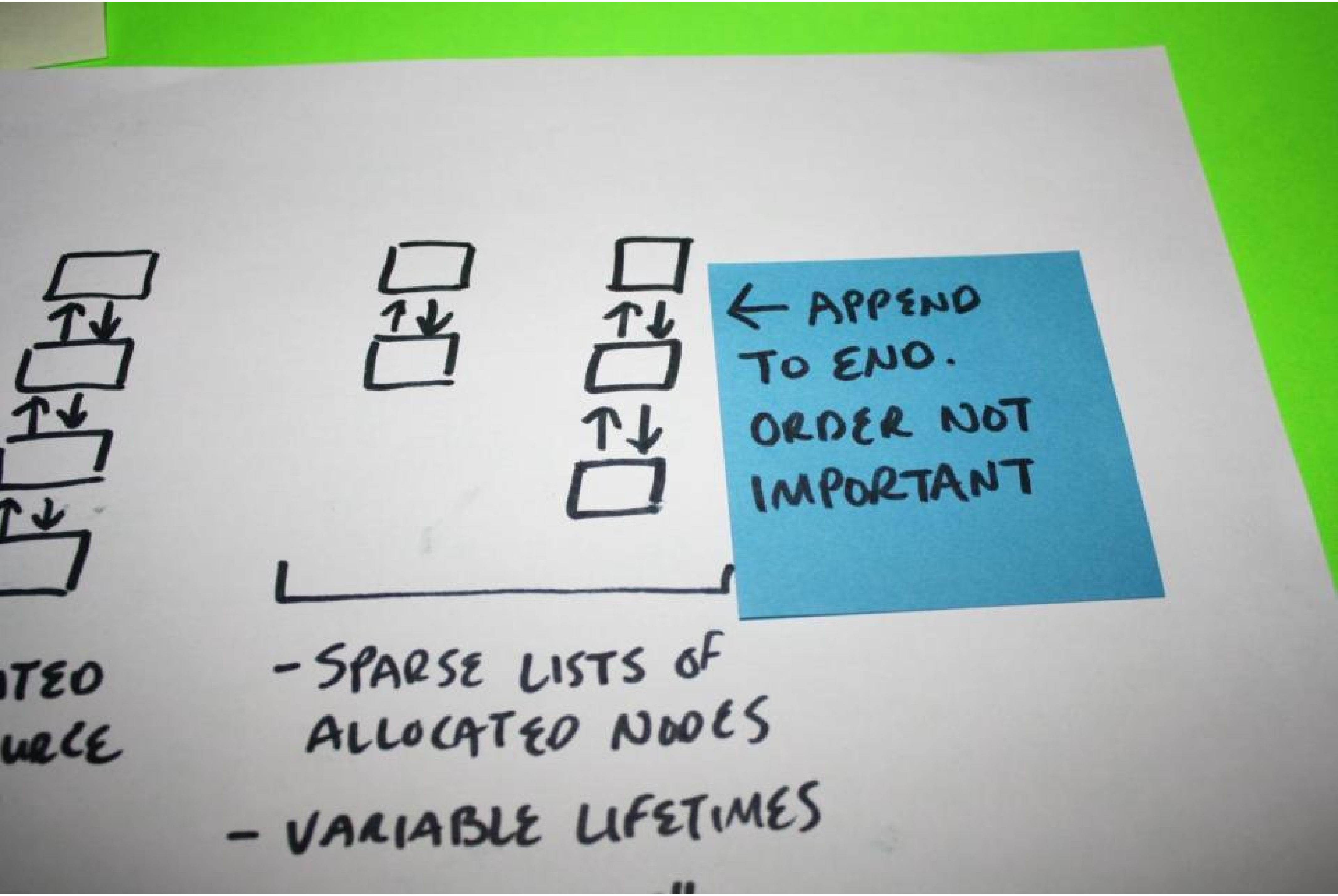
- SPARSE LISTS OF ALLOCATED NODES
- VARIABLE LIFETIMES
- DIFF "OWNERS", DIFF LISTS

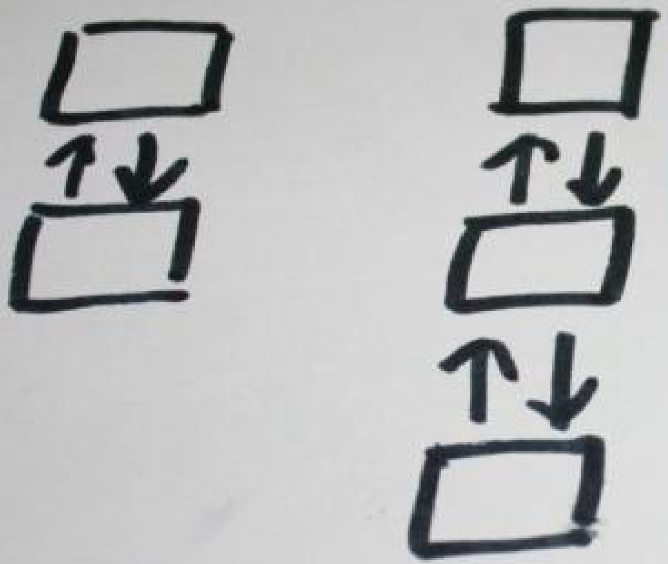
← As they're alloc'd, add to new "owner" lists.



- SPARSE LISTS OF ALLOCATED NODES
- VARIABLE LIFETIMES
- DIFF "OWNERS", DIFF LISTS

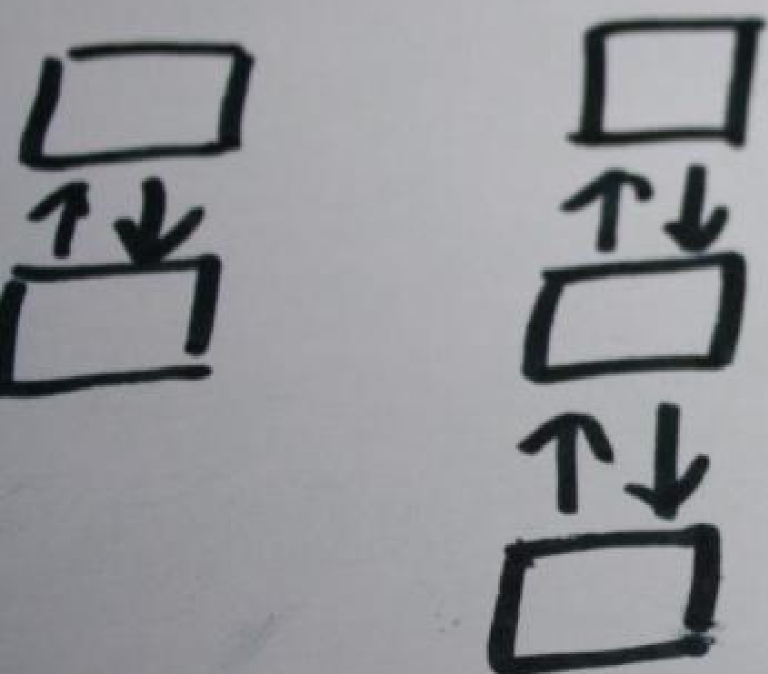
← When "deleted" remove from list, return to res. list.


← APPEND TO END. ORDER NOT IMPORTANT
- SPARSE LISTS OF ALLOCATED NODES
- VARIABLE LIFETIMES



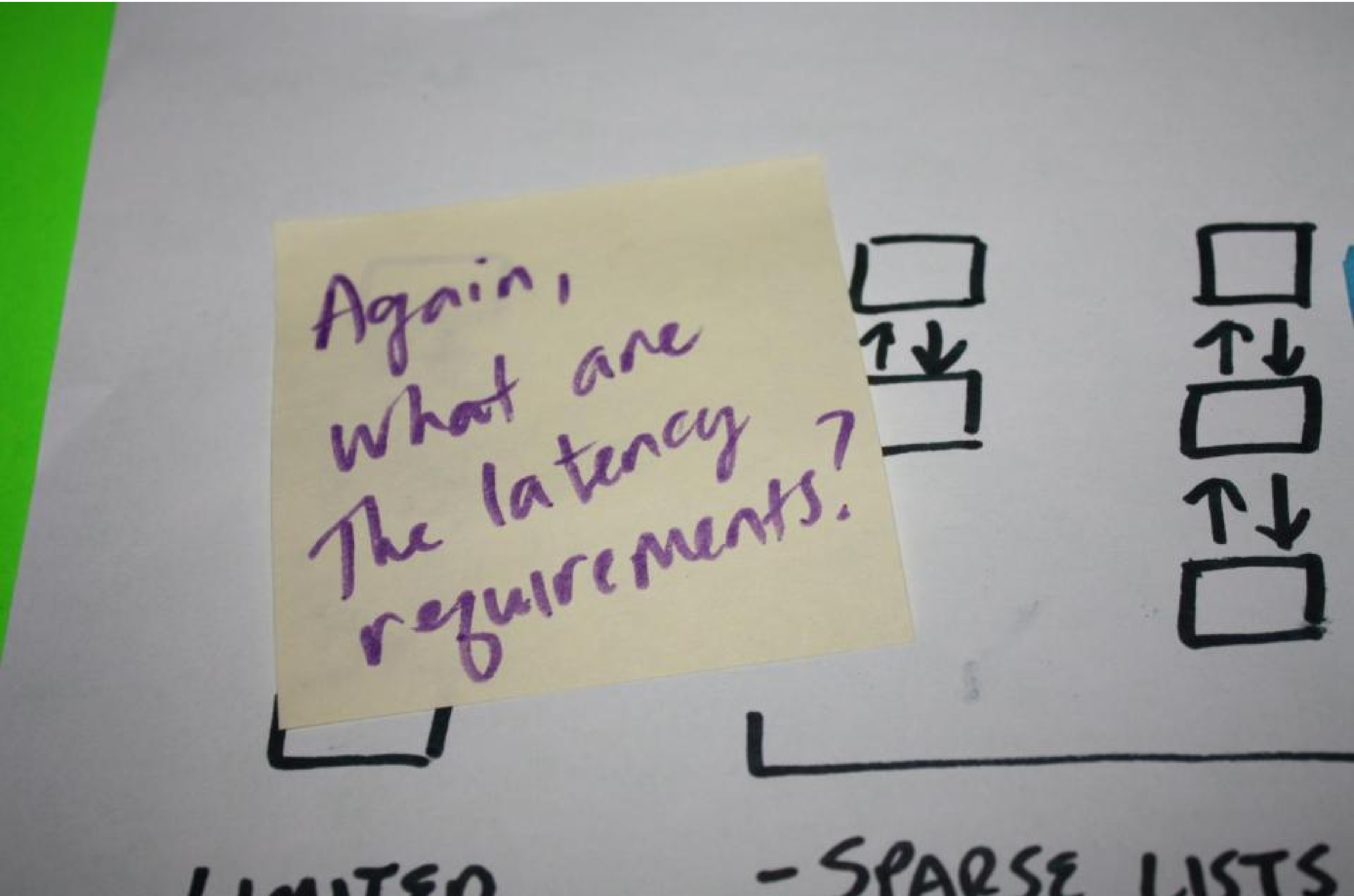
THIS IS THE EXPLICIT ORDERING RULE

- SPARSE LISTS OF ALLOCATED NODES



Note, implies
insert to
middle is
not needed.

LISTS OF


Again, what are The latency requirements?
- SPARSE LISTS

DATA FOR DIFFERENT ORDERING RULES WILL BE DIFFERENT.

DATA for different latency rules will be different.

Returning to the Question

Which answers/context does this structure match?

```
struct Node
{
    Node*   next;
    Node*   prev;
    Packet* data;
};
```

Returning to the Question

Which answers/context does this structure match?

```
struct Node
{
    Node*   next;
    Node*   prev;
    Packet* data;
};
```

NONE. Each set of rules for concurrent ops and requirements for latency, etc. would require a completely different struct.

So how to define what the data would be?

Begin at
the beginning

Hardware is the beginning.

Concurrency problems can't be abstracted from how hardware works.

WHAT IS THE "ASSEMBLY" OF CONCURRENCY?

i.e. What are the basic "primitives" to build concurrency solutions with?

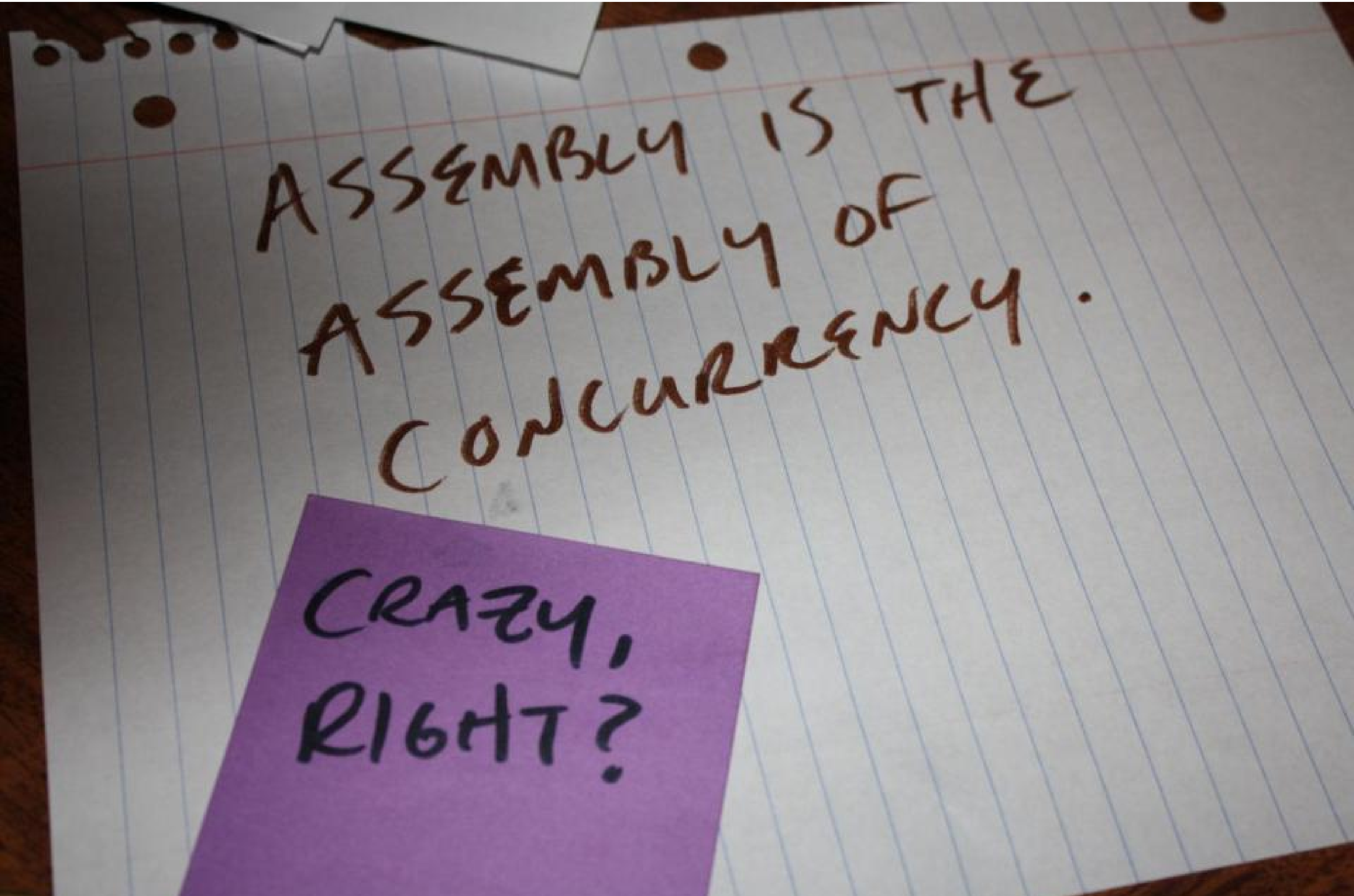
Mutexes?

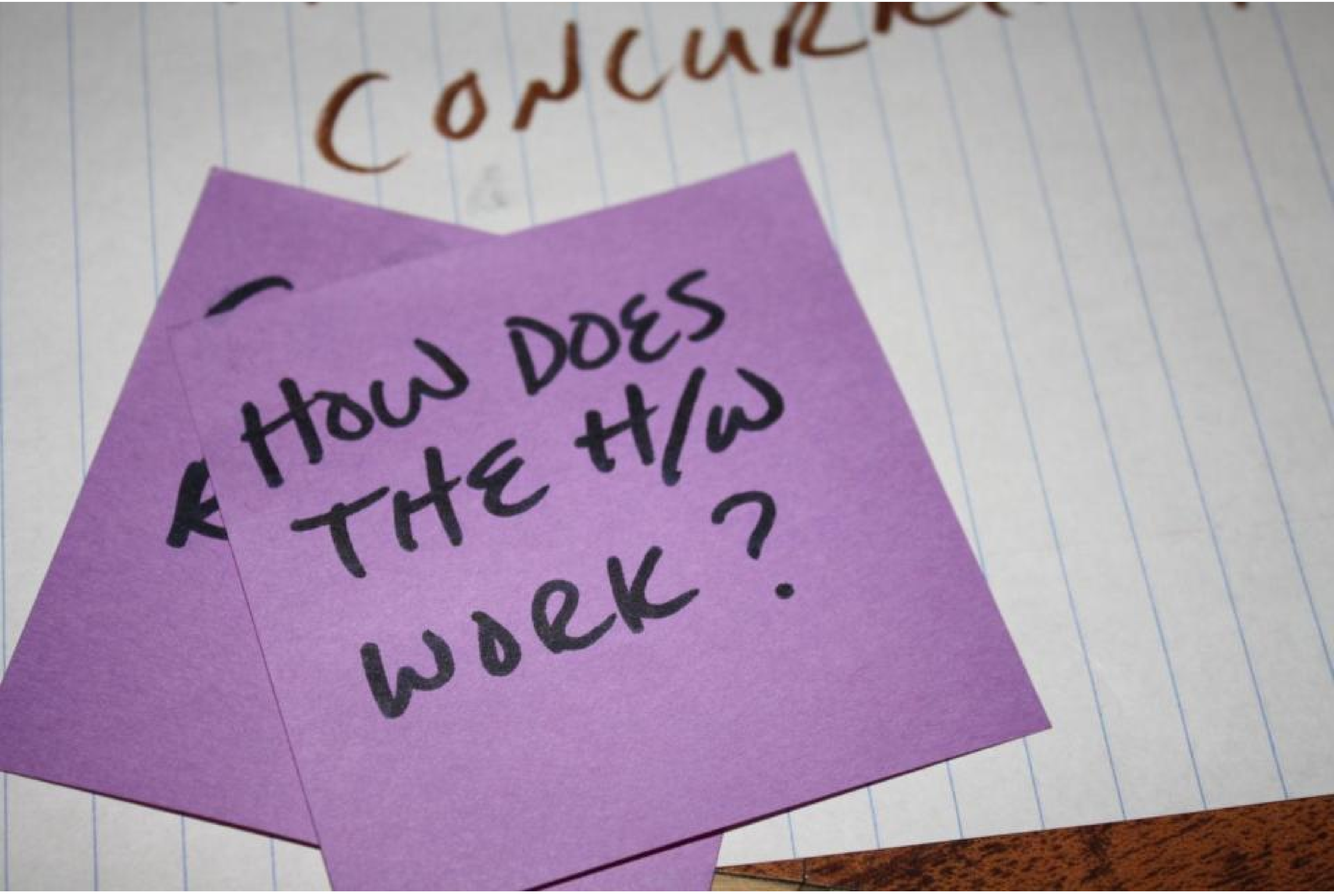
Semaphores?

Mailboxes?

Events?

ASSEMBLY IS THE ASSEMBLY OF CONCURRENCY.


ASSEMBLY IS THE
ASSEMBLY OF
CONCURRENCY.
CRAZY,
RIGHT?


HOW DOES
THE H/W
WORK?

ATOMIC READ
ATOMIC WRITE

HOW DO THEY WORK
ON THE TARGET H/W?

Concurrency starts with atomic transaction

#1 Practical Take-Away:

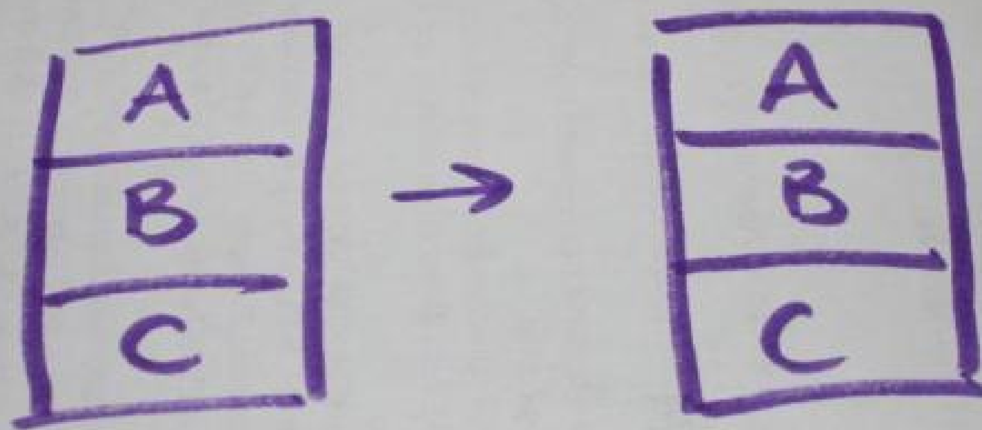
Know how to do lock-free atomic transaction on your h/w.

#1 Practical Take-Away:

Know how to do lock-free atomic transaction on your h/w.

- The fundamental data operation.
- Lock-free techniques built on top of this.

IN-ORDER STORES



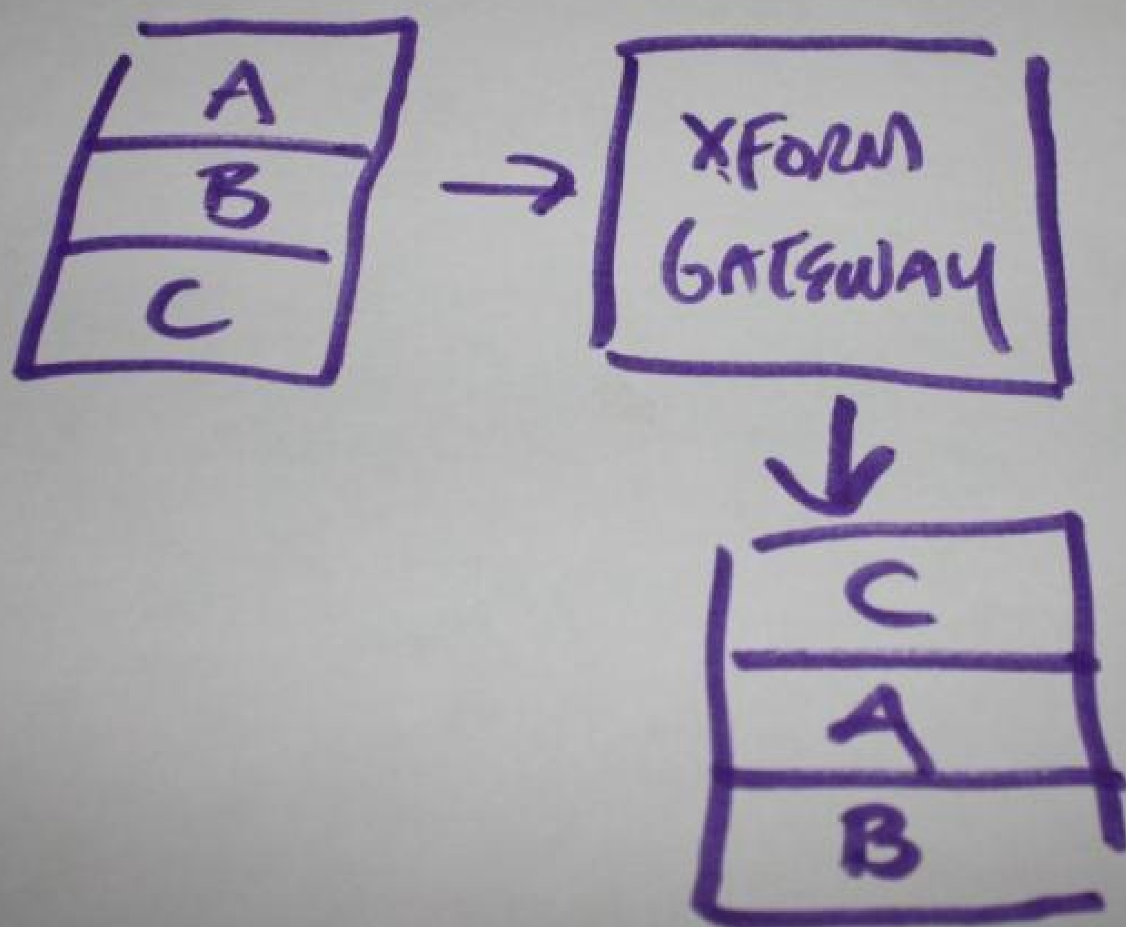
FIFO, NO CHANGE

IN-ORDER
PROCESSORS HAVE
IMPLICIT SYNC PT
EACH INSTRUCTION!

IN-ORDER
PROCESSORS HAVE
IMPLICIT SYNC PT
EACH INSTRUCTION!

↑
HERE USE
PRE-EXISTING
SYNC PTS.

OUT-OF-ORDER STORES



OUT-OF-ORDER
PROCESSORS PROVIDE
ORDERING PRIMITIVES
(e.g. FENCE)

OUT-OF-ORDER PROCESSORS PROVIDE ORDERING PRIMITIVES (e.g. FENCE)

BUT, MAY BE EVEN CHEAPER OPTION!

OCESSORS PROVI

ERING PRIMITIVES

(e.g. FENCE)

IF CAN
SUPPORT
HIGHER
LATENCY,
AND...

ESSORS
ING PRIMITIVES
(e.g. FENCE)

HIGHER-
LEVEL SYNC
PT ALREADY
EXISTS.

Note

In-order and out-of-order refers to load/store unit.

AKA weakly-ordered loads/stores
AKA load/store re-ordering

e.g.
SPU is in-order processor,
but MFC on SPU is not (out of order DMAs)

ALSO NOTE
LANGUAGE/COMPILER

Compiler/optmizer re-orders
instructions by definition!

PROG. MUST force
order!

Knowing h/w allows adding minimal sync points.

And..

GOOD CONCURRENCY DOESN'T ADD UNNEEDED SYNC POINTS.

GOOD CONCURRENCY DOESN'T ADD UNNEEDED SYNC POINTS.

BUT SHOULD DESIGN AROUND PRE-EXISTING ONES.

GOOD CONCURRENCY
DOESN'T ADD UNNEEDED
SYNC POINTS.

OR USE
CHEAPEST
ONES
AVAILABLE

Okay,
So now what needs to be defined BEFORE we can even begin to define the data?

NEED TO ANSWER:

- HOW WILL DATA BE TRANSFORMED?

- WHAT ARE THE CONSTRAINTS?

NEED TO ANSWER:

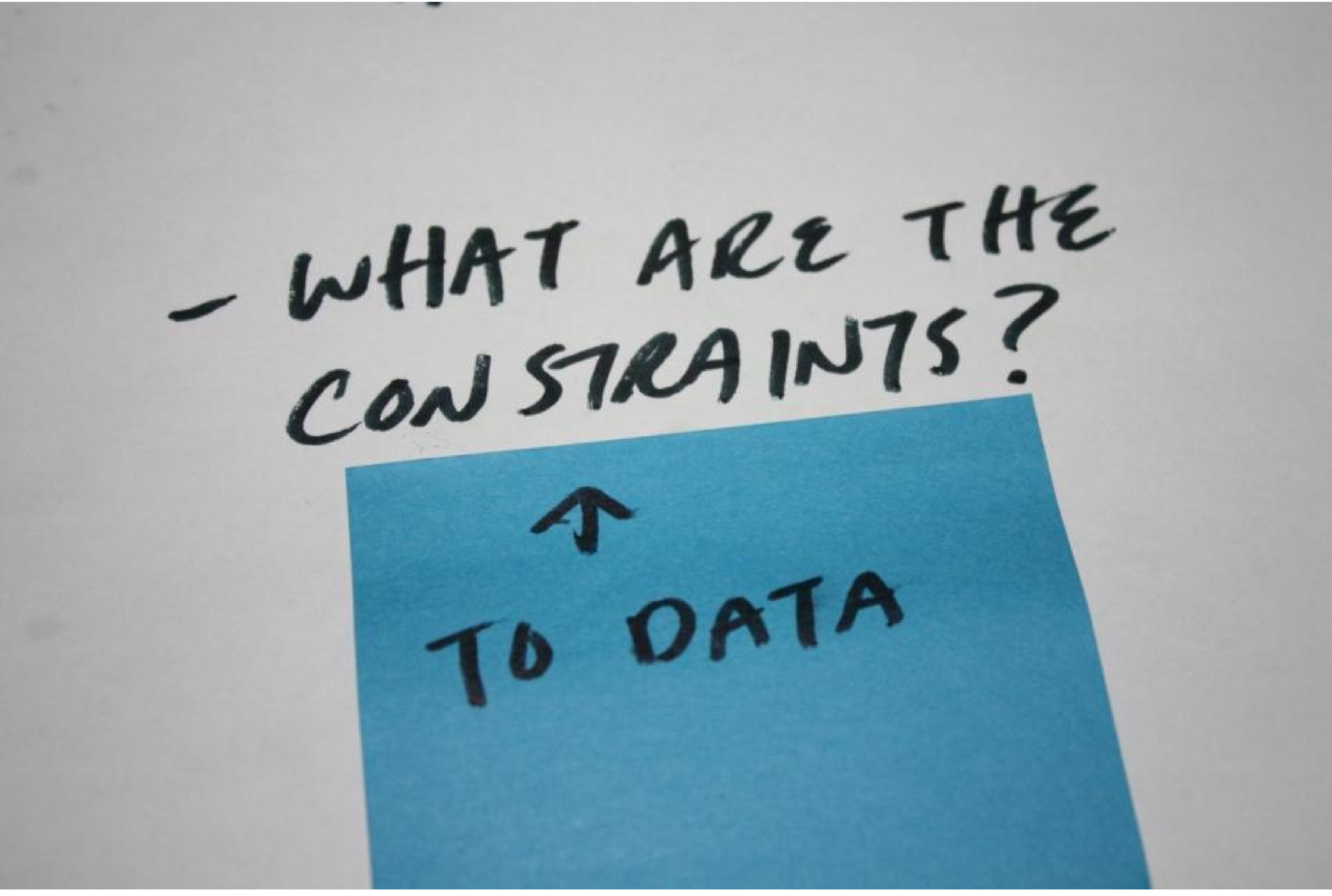
- HOW WILL DATA BE TRANSFORMED?

↑

I.E.
OPERATIONS

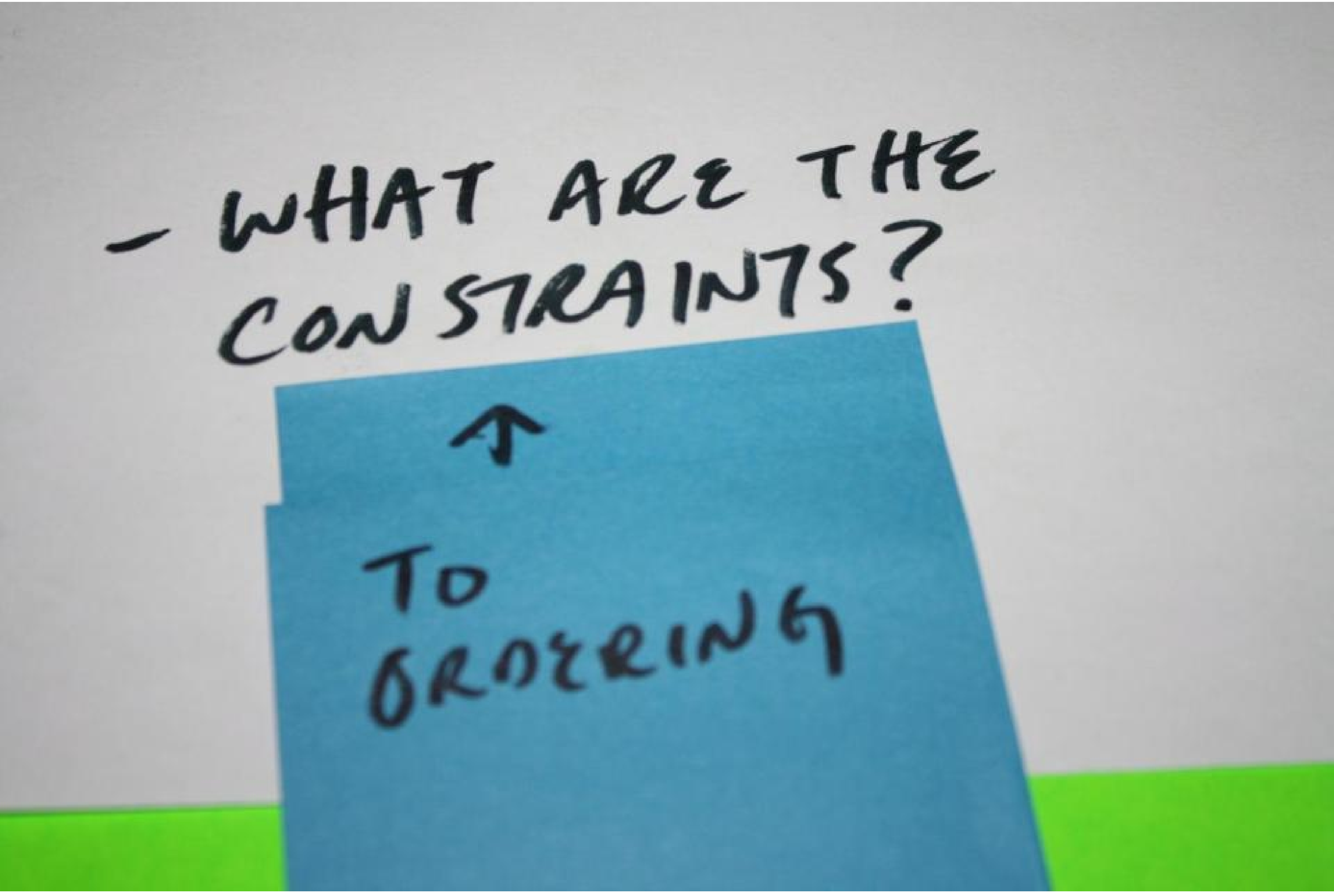
- WH

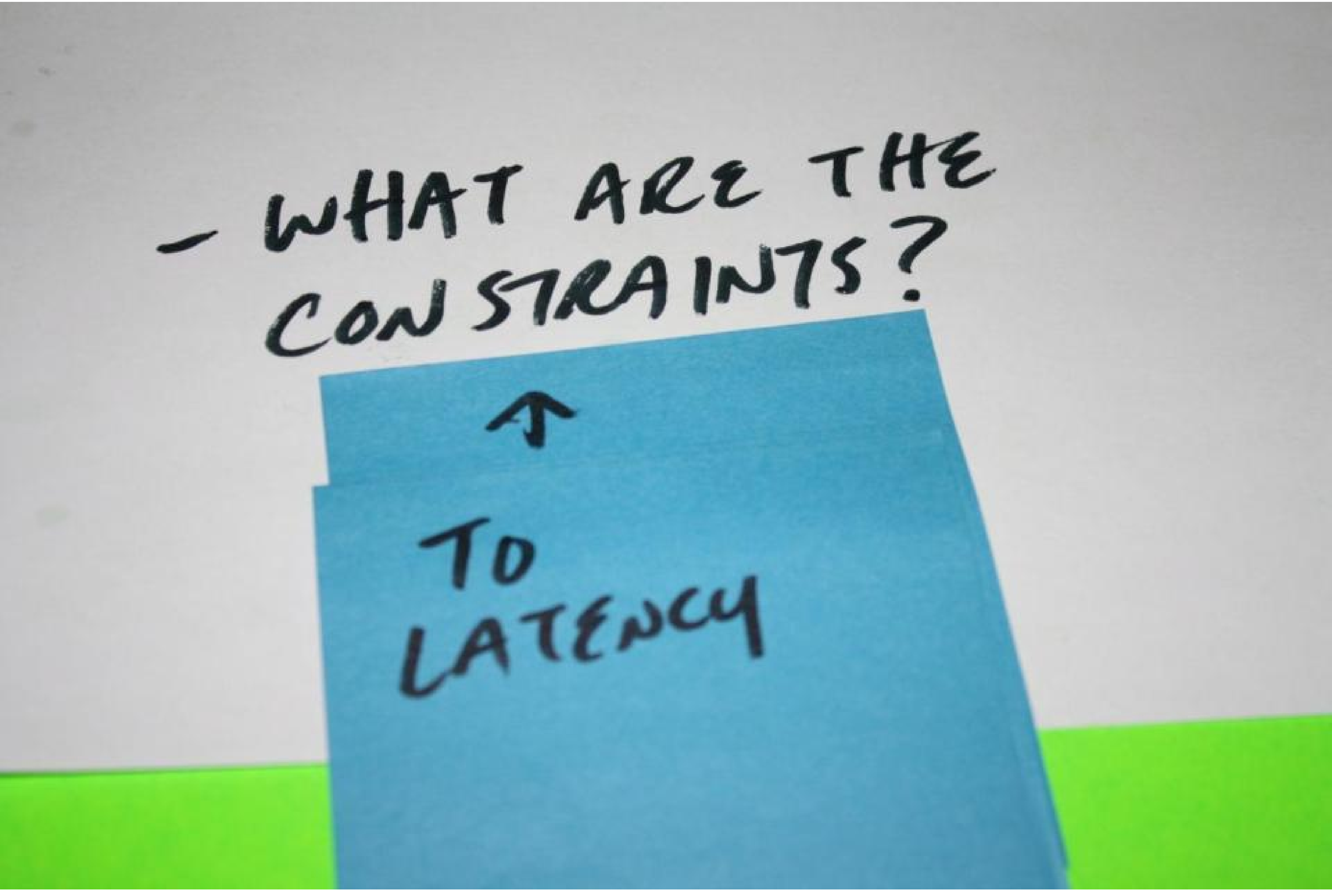
CON


- WHAT ARE THE CONSTRAINTS?
TO DATA

– WHAT ARE THE CONSTRAINTS?

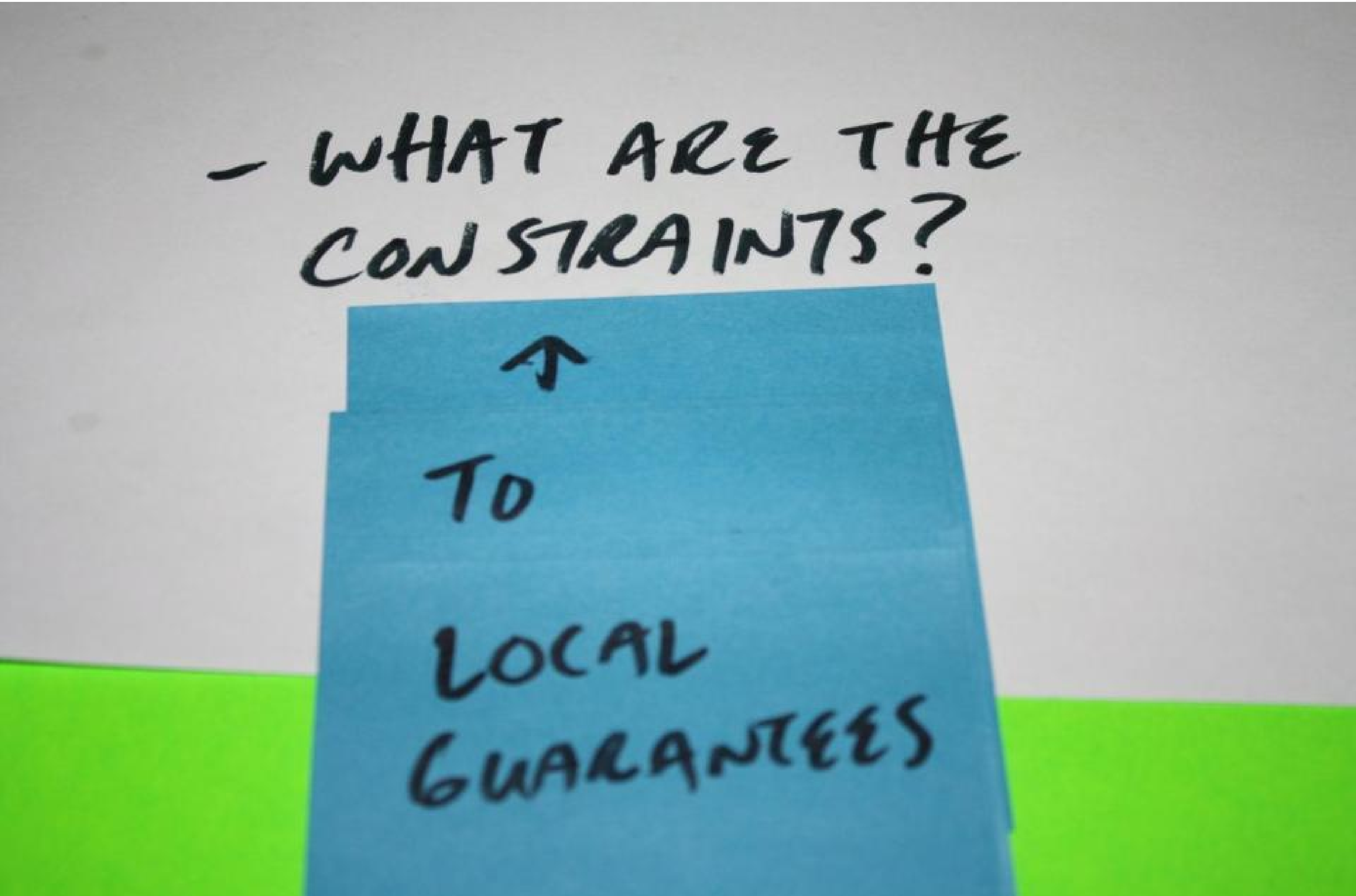
TO TRANSFORMATION


- WHAT ARE THE CONSTRAINTS?
TO ORDERING


- WHAT ARE THE CONSTRAINTS?
TO LATENCY

- WHAT ARE THE CONSTRAINTS?

TO GLOBAL GUARANTEES


- WHAT ARE THE CONSTRAINTS?
TO
LOCAL GUARANTEES

Do that and you're on your way.

Next set of talks will apply these lessons to optimizing data for specific examples.

But the "optimized" part will introduce something new...

Here's a little teaser...

THE SECRET

of OPTIMIZED

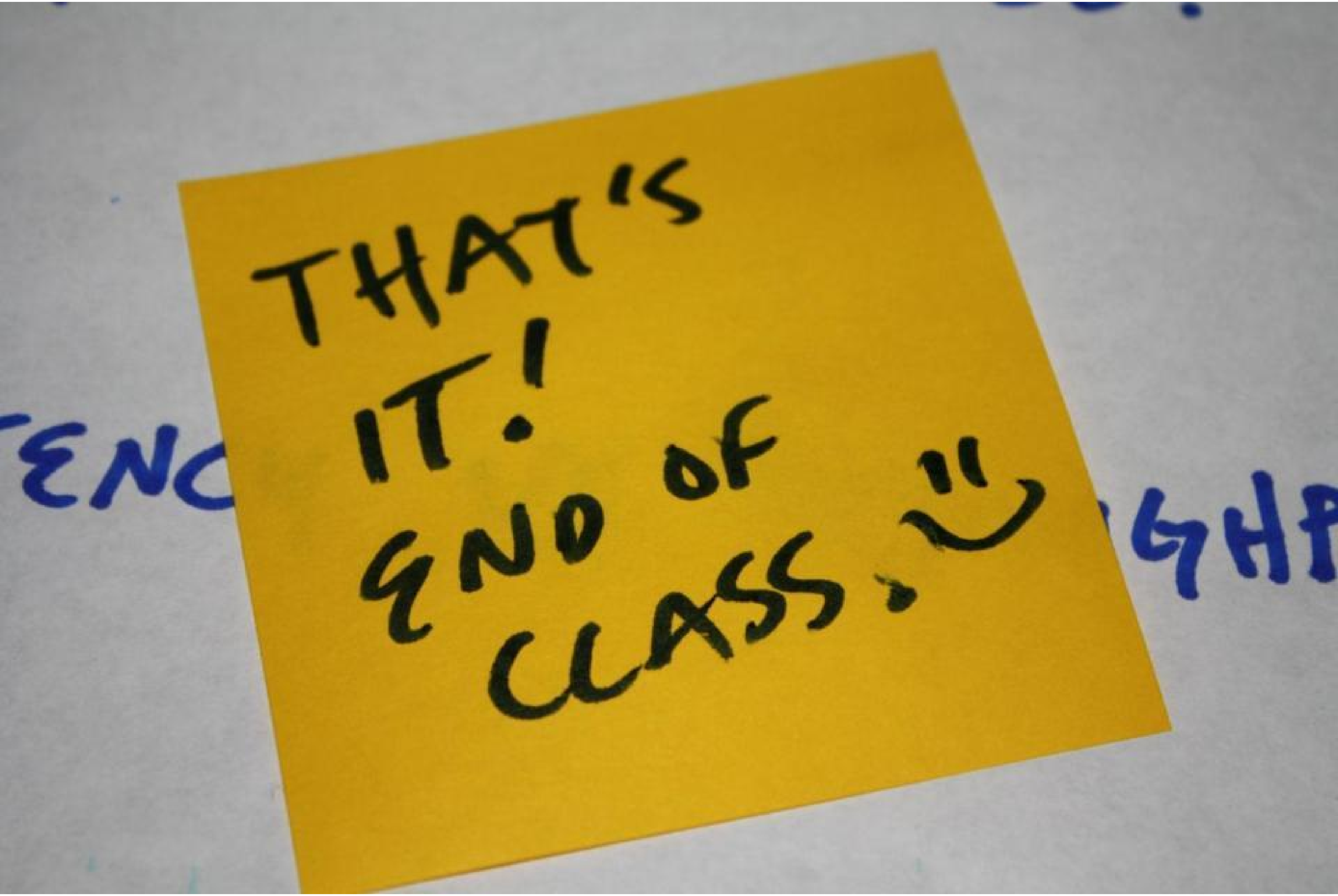
concurrent design

is...

DELAY*

* kind of ironic, right?

THE SOONER YOU NEED
SOME DATA, THE SLOWER
THE SYSTEM WILL BE.


THAT'S
IT!
END OF
CLASS.

Thanks!

Great feedback from Bjoern Knafla (@bjoernknafla on twitter). Plus he came up with the name for the presentation.

Also thanks for feedback: @MarcoSalvi, @rickmolloy

Also thanks Rob Wyatt for feedback - and for suggestions on even more fun problems and complex cases that we could cover to make the presentation even longer next time.