REAL WORLD WEB SCALE INFORMATION RETRIEVAL

featuring Mike Malone

A NARRATIVE

Exposition

It all started with an idea... followed by an even better idea.

Conflict

Consistency and availability, fault tolerance and redundancy. Oh my!

Rising Action

A spatial database!? Secondary conflicts, obstacles, and frustrations!

Climax

A turning point that marks a change in the protagonist's affairs.

Conclusion..?

Sequel likely!

THE LEAD

MIKE MALONE
INFRASTRUCTURE ENGINEER

mike@simplegeo.com
@mjmalone

For the last 10 months I've been working on a web-scale spatial database that lives at the core of SimpleGeo's infrastructure.

SIMPLEGEO

We originally began as a mobile gaming startup, but quickly discovered that the location services and infrastructure needed to support our ideas didn't exist. So we took matters into our own hands and began building it ourselves.

Matt Galligan
CEO & co-founder

Joe Stump
CTO & co-founder

AS PROTAGONIST

My goal is to summarize the last ten months of R&D work at SimpleGeo in an hour.

IN REALITY...

I'm leaving out lots of the details. Some of them are interesting. And that's sad.

REQUIREMENTS

We like all of our data and want to keep it around. Therefore, our database must be:

- HIGHLY AVAILABLE
- FAULT TOLERANT
- DECENTRALIZED
- HORIZONTALLY SCALABLE
- OPERATIONALLY SIMPLE

CANDIDATES

THE TRANSACTIONAL RELATIONAL DATABASES

- They're theoretically pure, well understood, and mostly standardized behind a relatively clean abstraction
- They provide robust contracts that make it easy to reason about the structure and nature of the data they contain
- They're battle tested, hardened, robust, durable, etc.

OTHER STRUCTURED STORAGE OPTIONS

- Plain I see you, western youths, see you tramping with the foremost, Pioneers! O pioneers!

COLORADO MEN ARE WE

WE CHOSE "OTHER STRUCTURED STORAGE"

- Traditional transactional databases **aren't high availability** or fault tolerant without external management
- Traditional transactional databases **are centralized** without external management
- Traditional transactional databases **can't scale horizontally** without external management
- External management of traditional transactional databases is difficult to build, domain specific, and **operationally complex**

IN SHORT, THE TRADITIONAL TRANSACTIONAL RDBMS SOLUTIONS DIDN'T MEET OUR REQUIREMENTS...

THEY SIMPLY CAN'T.

IF SOMEONE CLAIMS TO HAVE WRITTEN A HIGH AVAILABILITY HORIZONTALLY SCALABLE APP ON A TRADITIONAL RDBMS, IT'S BECAUSE THEY WROTE THEIR OWN DBMS ON TOP OF IT.

THAT'S NOT NECESSARILY A BAD THING THOUGH.

ACID IN 30 SECONDS

These terms are not formally defined - they're a framework, not mathematical axioms

ATOMICITY

Either all of a transaction's actions are visible to another transaction, or none are

CONSISTENCY

Application-specific constraints must be met for transaction to succeed

ISOLATION

Two concurrent transactions will not see one another's transactions while "in flight"

DURABILITY

The updates made to the database in a committed transaction will be visible to future transactions

ACID HELPS

ACID is a sort-of-formal contract that makes it easy to reason about your data, and that's **good**

IT DOES SOMETHING HARD FOR YOU

- With ACID, you're guaranteed to maintain a persistent global state as long as you've defined proper constraints and your logical transactions result in a valid system state

ACID HURTS

Certain aspects of ACID encourage (require?) implementors to do “**bad** things”

Unfortunately, ANSI SQL’s definition of isolation...

> relies in subtle ways on an assumption that a locking scheme is used for concurrency control, as opposed to an optimistic or multi-version concurrency scheme. This implies that the proposed semantics are ill-defined.

Joseph M. Hellerstein and Michael Stonebraker
Anatomy of a Database System

CAP THEOREM IN 30 SECONDS

At PODC 2000 Eric Brewer told us there were three desirable DB characteristics. But we can only have two.

CONSISTENCY

Every node in the system contains the same data (e.g., replicas are never out of date)

AVAILABILITY

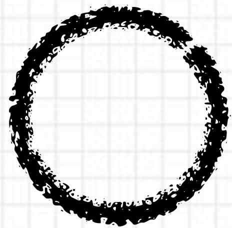
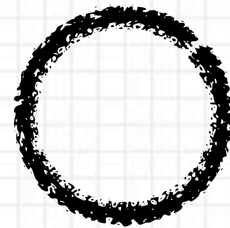
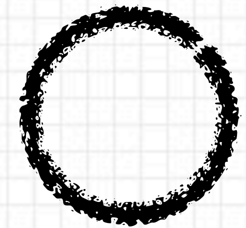
Every request to a non-failing node in the system returns a response

PARTITION TOLERANCE

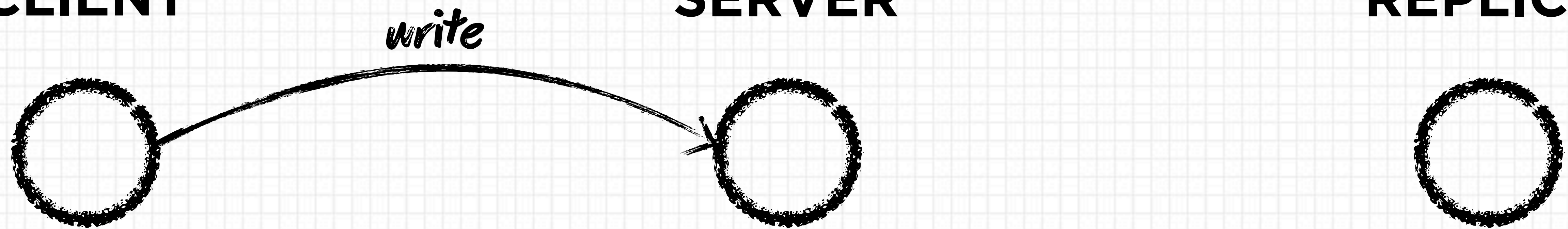
System properties (consistency and/or availability) hold even when the system is partitioned and data is lost

CAP THEOREM IN 30 SECONDS

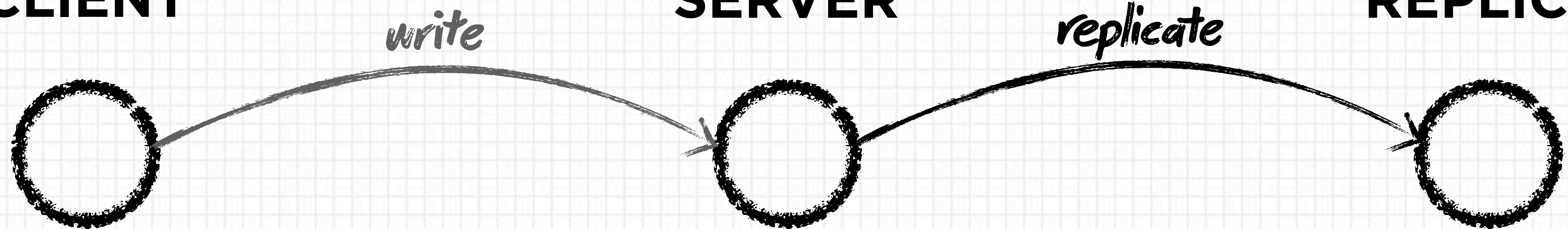
CLIENT **SERVER** **REPLICA**

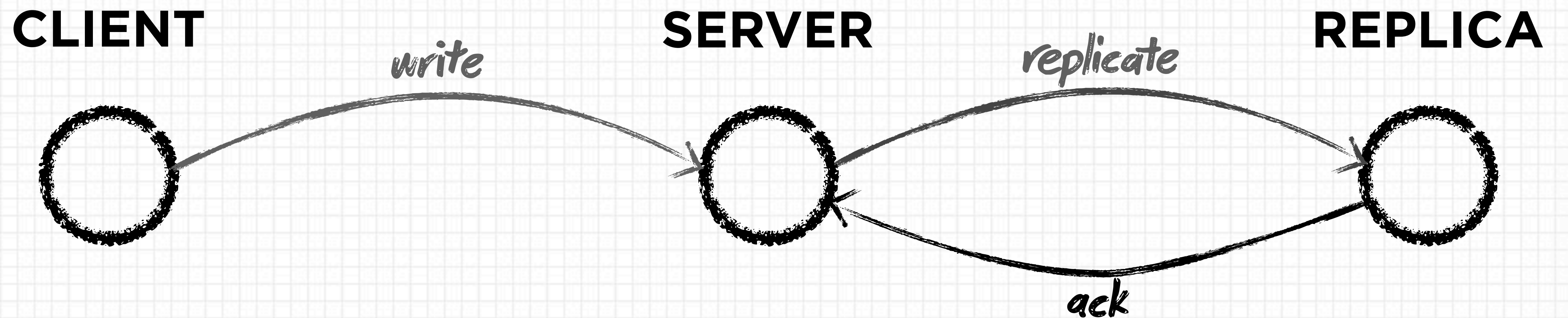
CAP THEOREM IN 30 SECONDS



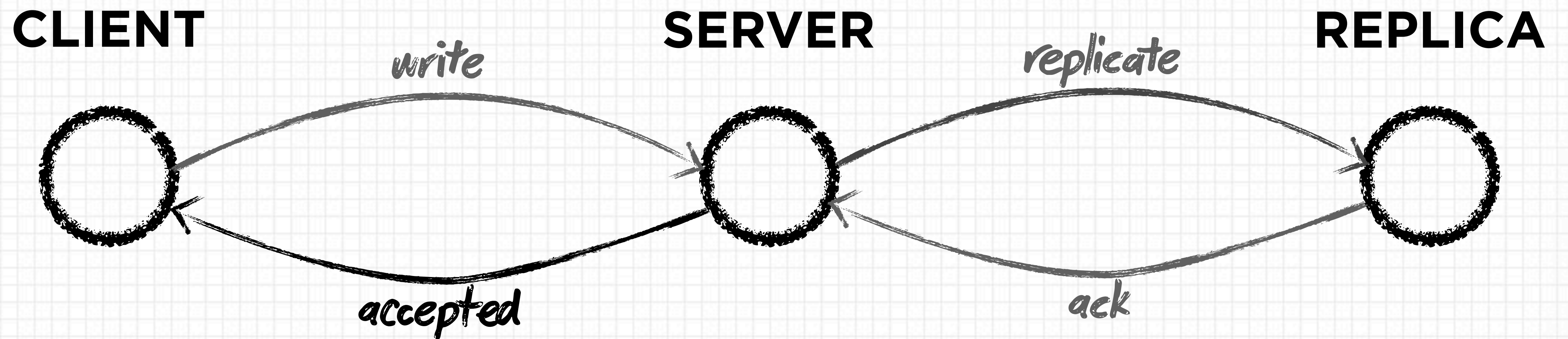
CAP THEOREM IN 30 SECONDS



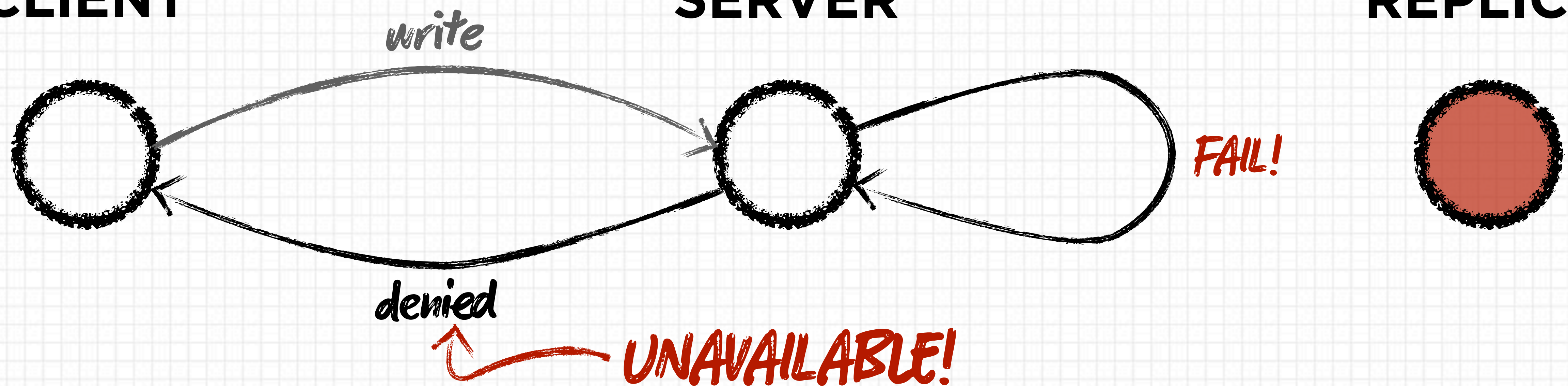
CAP THEOREM IN 30 SECONDS



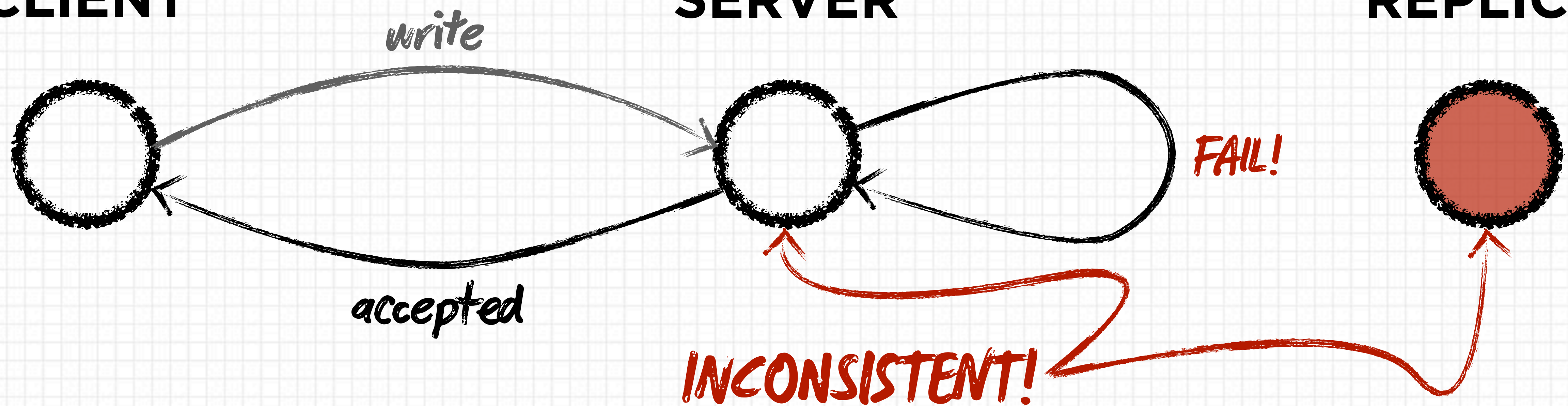
CAP THEOREM IN 30 SECONDS



CAP THEOREM IN 30 SECONDS



CAP THEOREM IN 30 SECONDS



IN A NUTSHELL

IT'S A QUESTION OF VALUES

- For **traditional** databases **CAP consistency** is the holy grail: it's maximized at the expense of availability and partition tolerance
- At scale, failures happen: when you're doing something a million times a second a one-in-a-million failure happens every second
- We're witnessing the birth of a **new religion**...
  - CAP consistency is a luxury that must be sacrificed at scale in order to maintain availability when faced with failures
- But let's not have a religious war - horses for courses

A SYSTEM IN ITS HAPPY STATE WILL BEHAVE "CONSISTENTLY!" IT'S WHAT HAPPENS DURING FAILURES THAT DIFFERS.

RISING ACTION

APACHE CASSANDRA

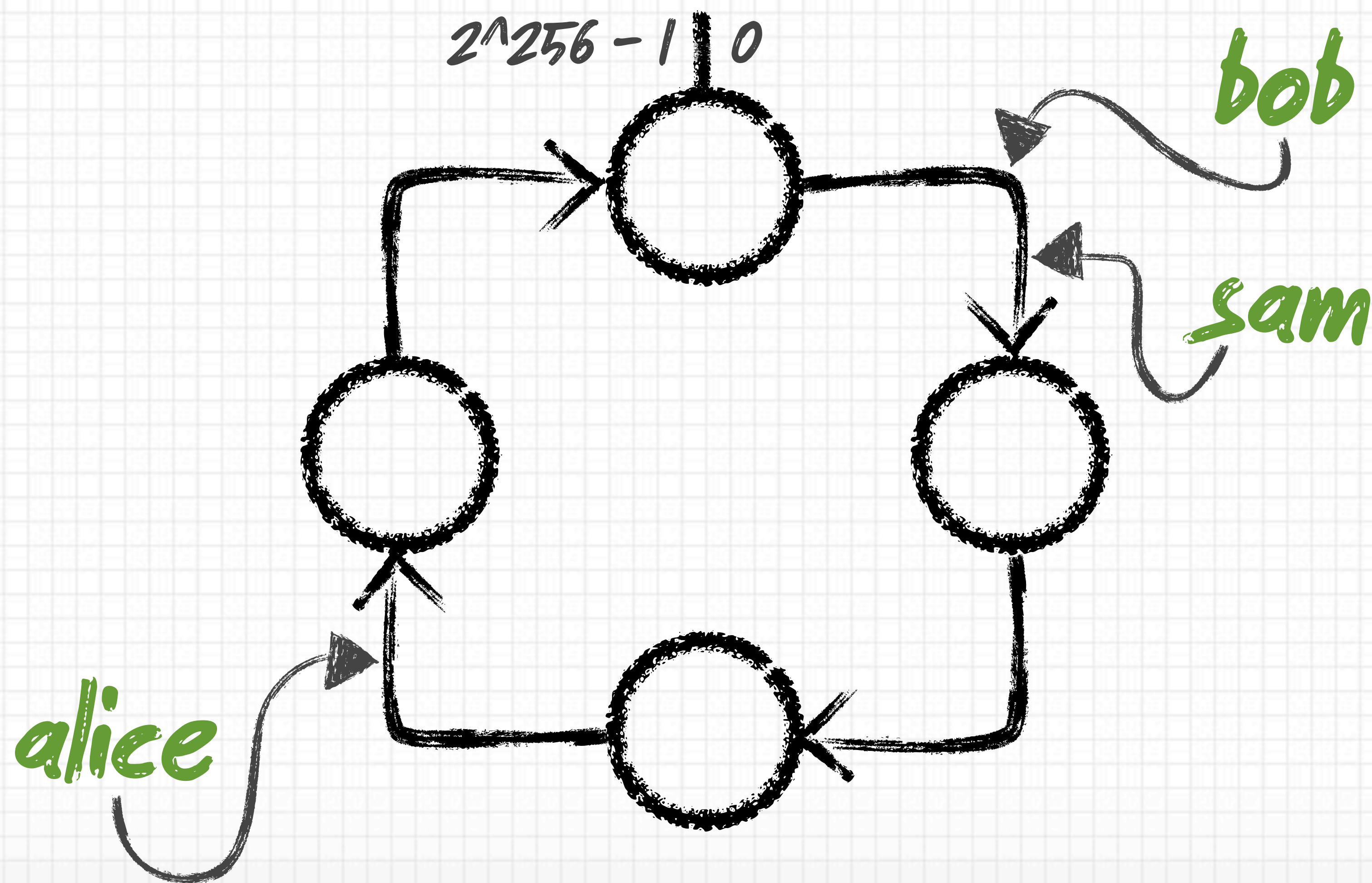
IT'S A GREAT DISTRIBUTED HASH TABLE

- Gossip provides **fault detection & tolerance** and **distributed** operations with **minimal operational overhead**
- Random hash-based partitioning provides **auto-scaling** and efficient online rebalancing
- Pluggable **replication** for multi-datacenter replication
- **Tunable consistency** allow us to adjust durability with the value of data being written
- As a pure **peer-to-peer** system operations are **decentralized** and the cluster can automatically heal after failures

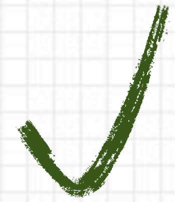
I LIKE SOFTWARE WITH ACADEMIC RIGOR AND I DON'T LIKE TO RUN SOFTWARE I DON'T UNDERSTAND

CASSANDRA IS NOT A SPATIAL DATABASE

DISTRIBUTED HASH TABLES INTENTIONALLY DESTROY DATA LOCALITY (BY HASHING) IN ORDER TO ACHIEVE GOOD BALANCE AND SCALE LINEARLY



HASH TABLE: SUPPORTED QUERIES


EXACT MATCH

RANGE

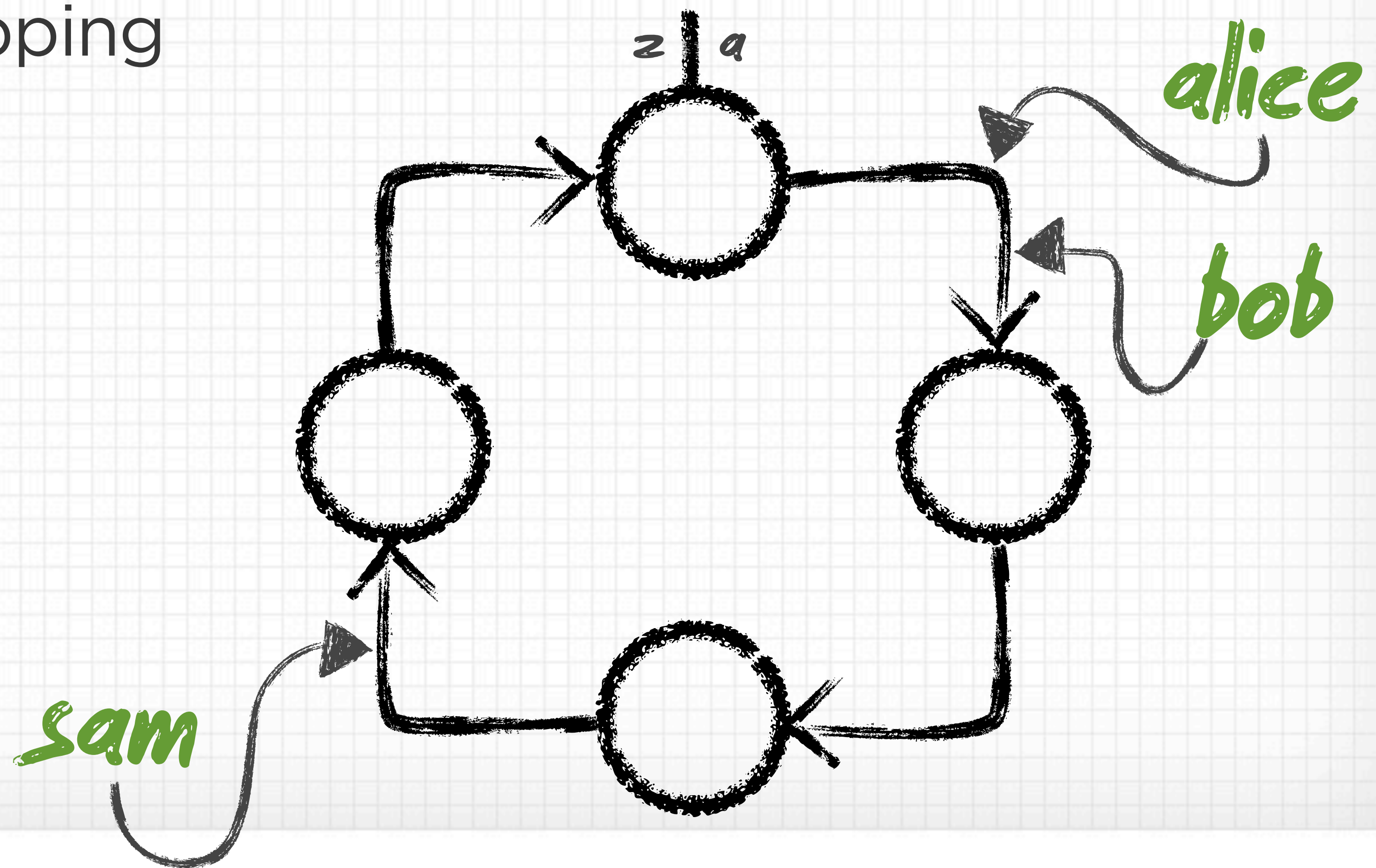
PROXIMITY

ANYTHING THAT'S NOT EXACT MATCH

THE ORDER PRESERVING PARTITIONER

CASSANDRA'S PARTITIONING STRATEGY IS PLUGGABLE

- The partitioner is responsible for mapping a key to a node
- The order preserving partitioner uses the natural ordering of the row keys for this mapping



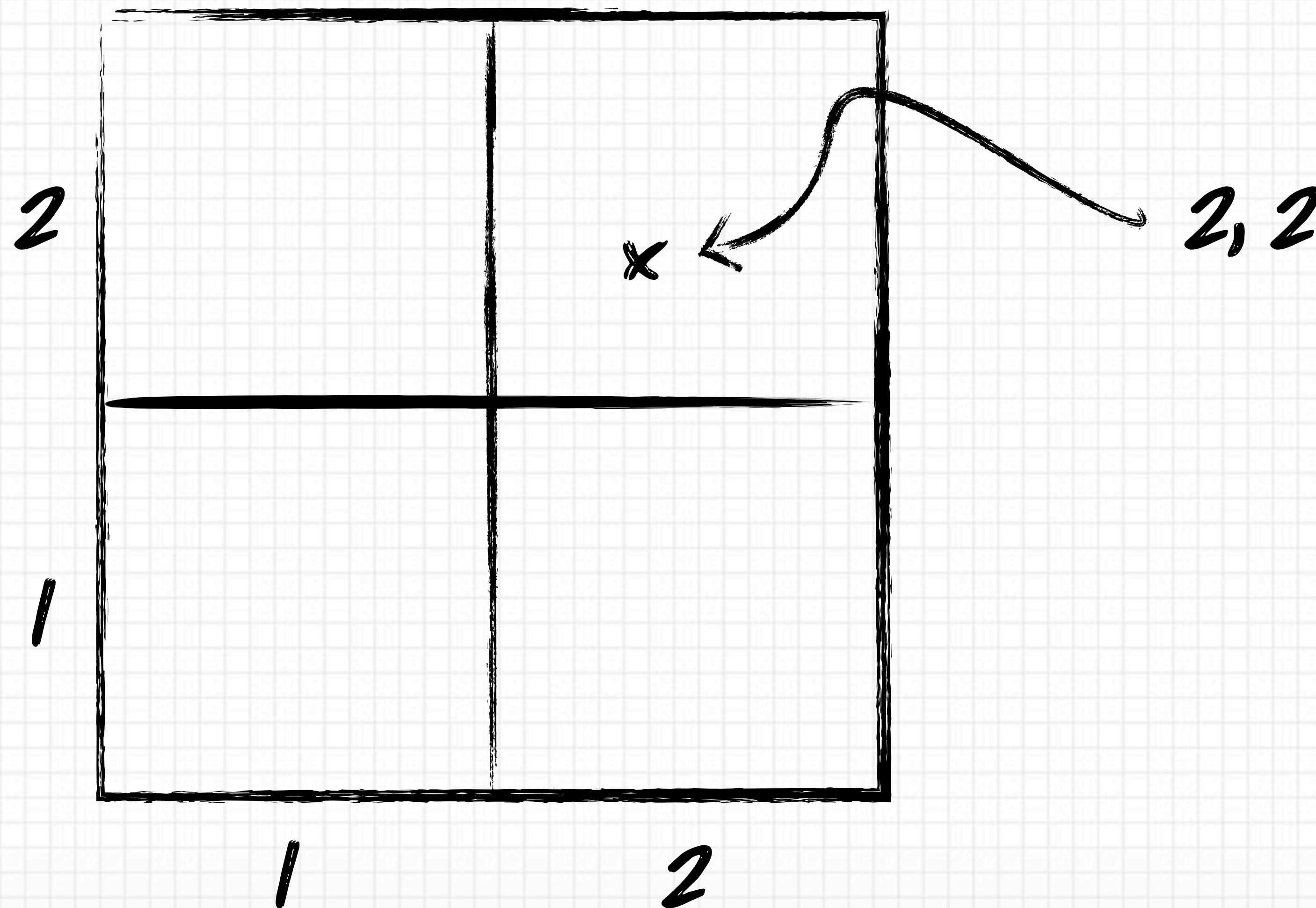
ORDER PRESERVING PARTITIONER

BECAUSE DATA IS ARRANGED ACCORDING TO ITS NATURAL ORDERING, YOU CAN PERFORM RANGE QUERIES ON A *SINGLE DIMENSION*

- ✓ **EXACT MATCH**
- ✓ **RANGE**
- ? **PROXIMITY**

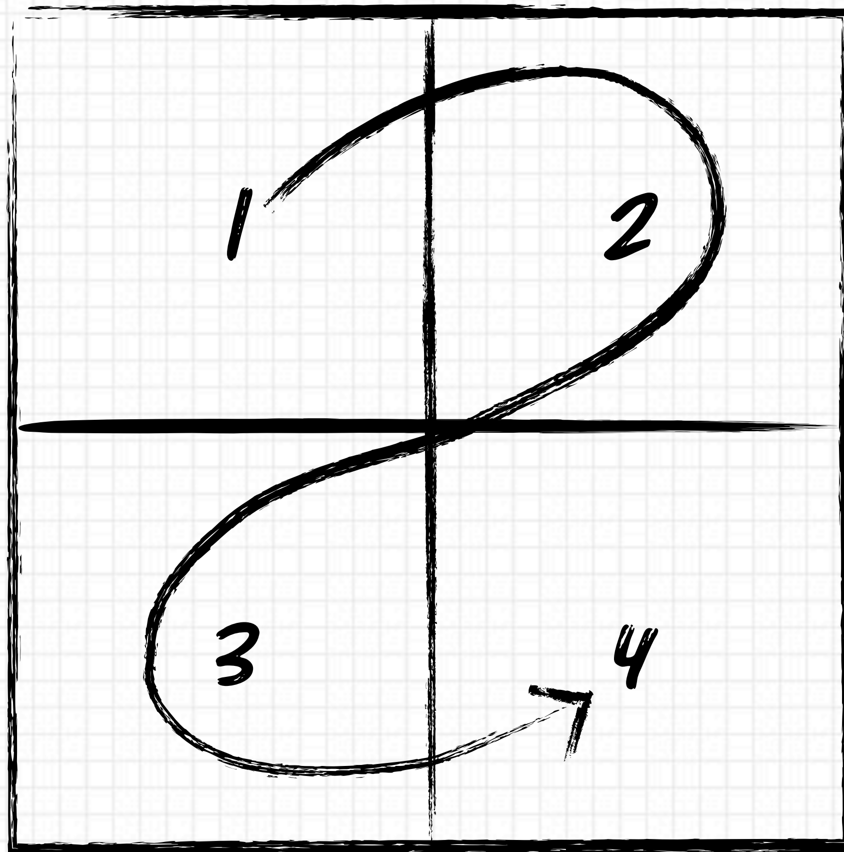
DIMENSIONALITY REDUCTION SPACE-FILLING CURVES

SPATIAL DATA IS INHERENTLY MULTIDIMENSIONAL...



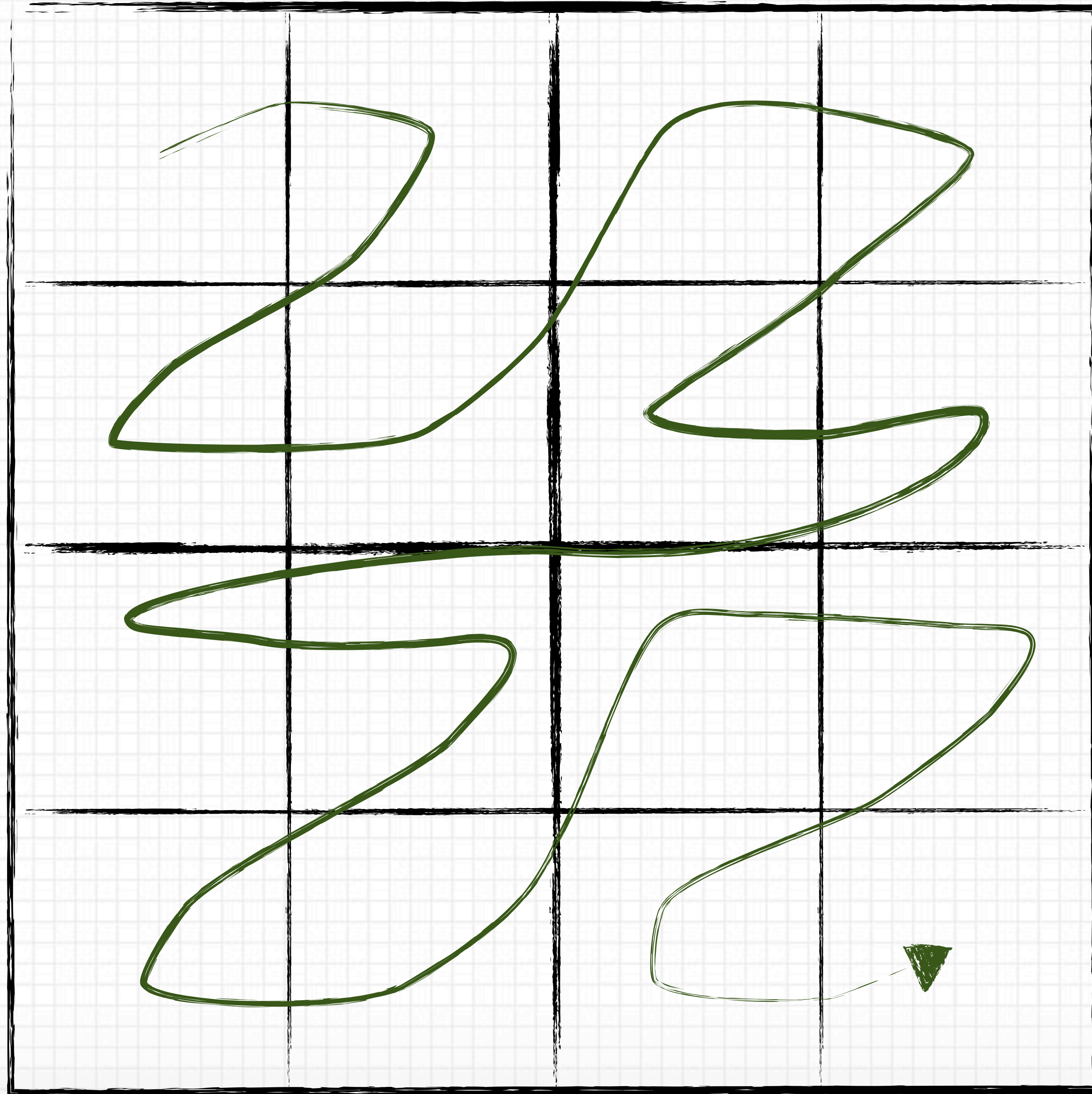
DIMENSIONALITY REDUCTION
SPACE-FILLING CURVES

SPATIAL DATA IS INHERENTLY MULTIDIMENSIONAL... OR IS IT!?



SPOILER ALERT: IT IS

Z-CURVE SECOND ITERATION

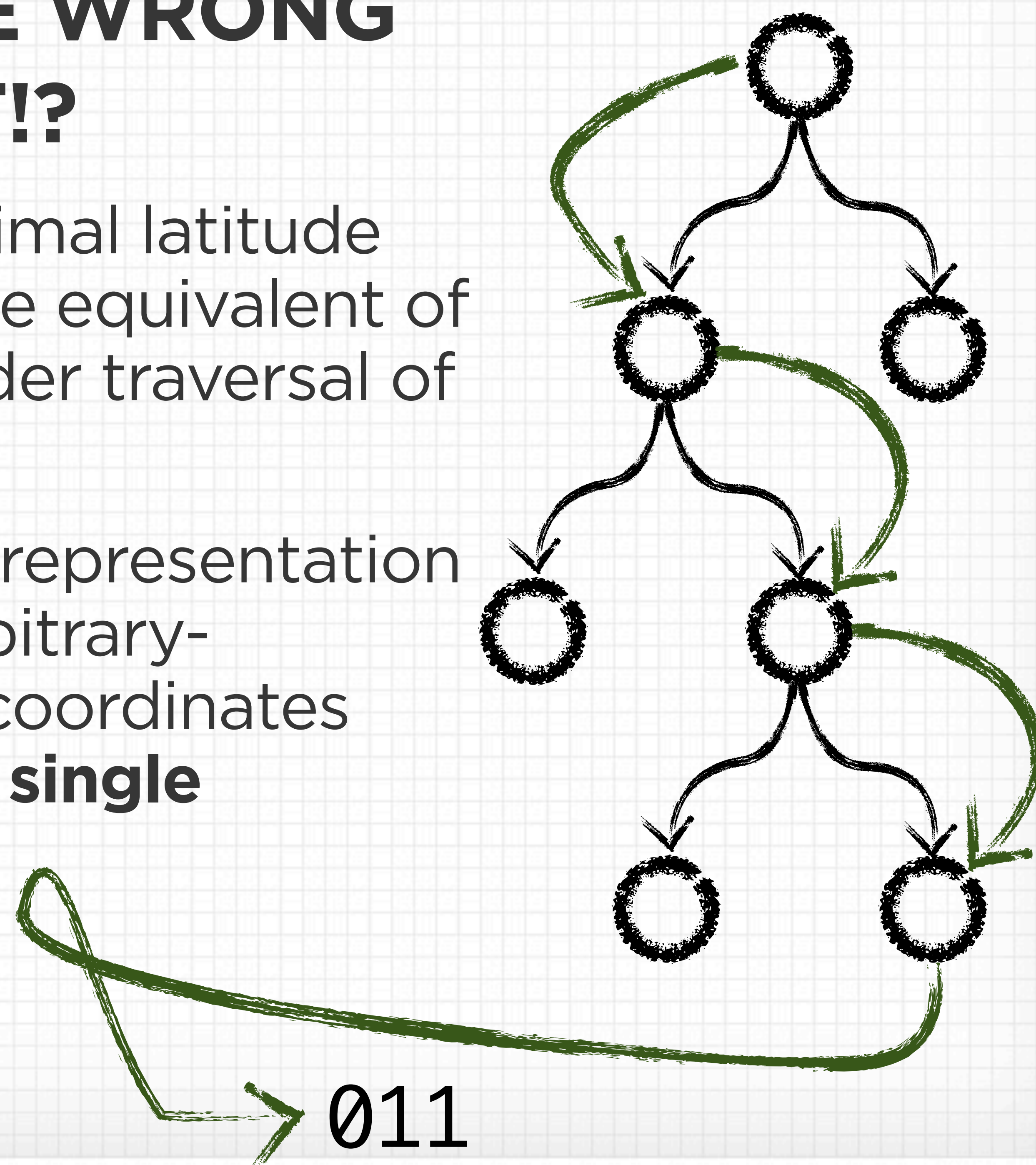


GEOHASH

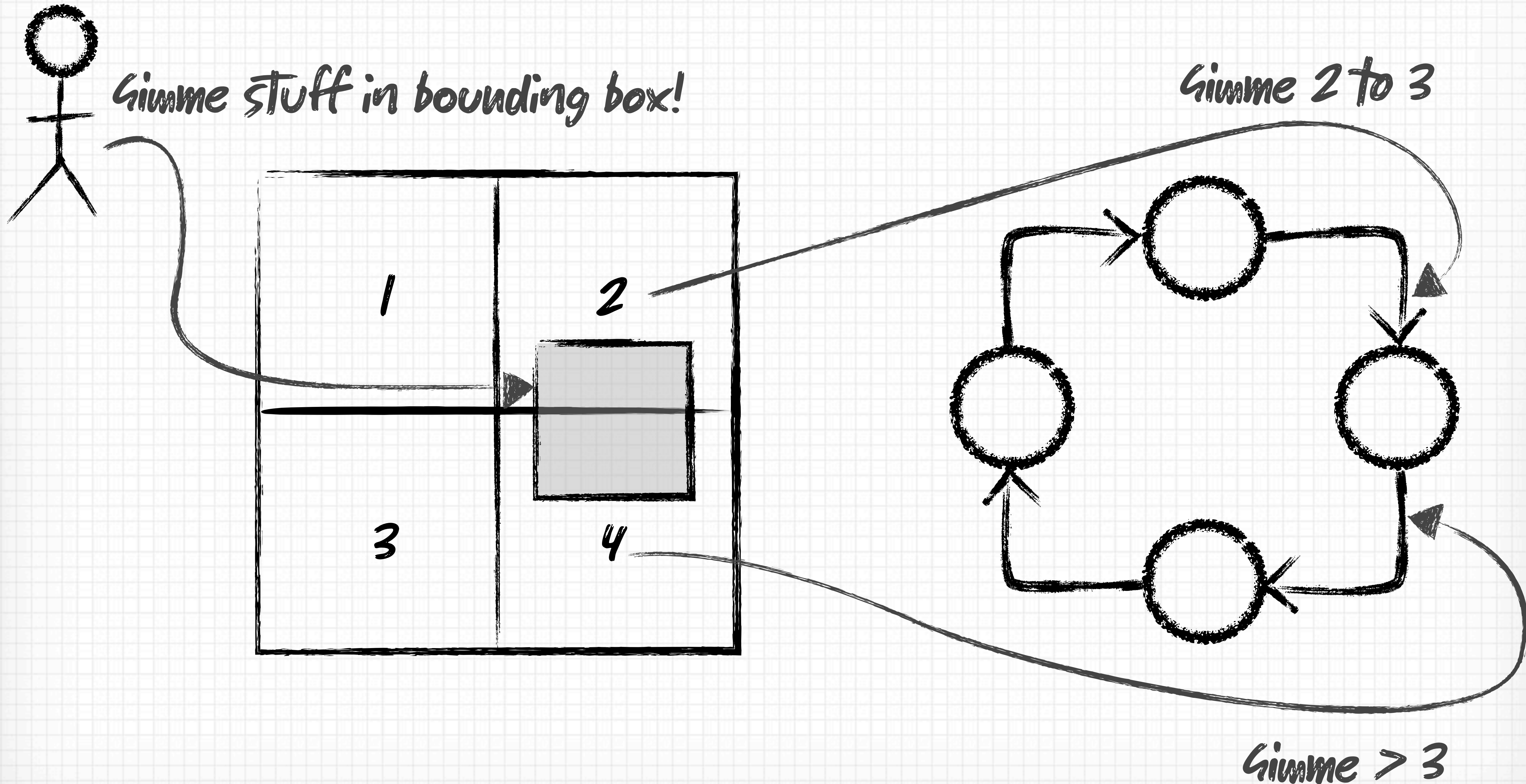
HOW CAN SOMETHING BE WRONG WHEN IT FEELS SO RIGHT!?

- By interleaving the bits of a decimal latitude and longitude you're left with the equivalent of a binary encoding of the pre-order traversal of a tree (no joke)
- By base32 encoding this binary representation you're left with a convenient, arbitrary-precision representation of the coordinates that **sorts lexicographically** in a **single dimension**

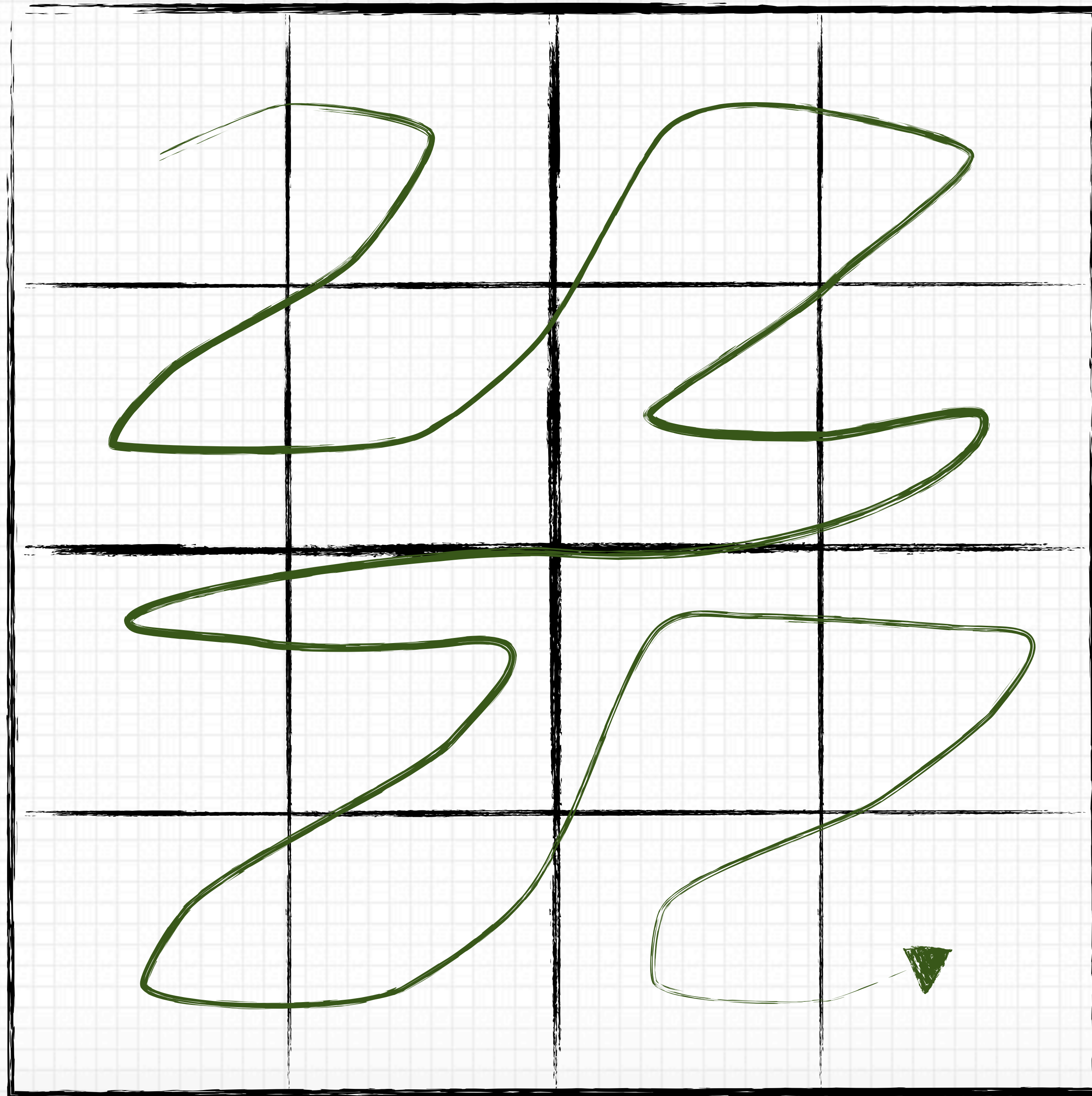
THAT'S SO COOL!



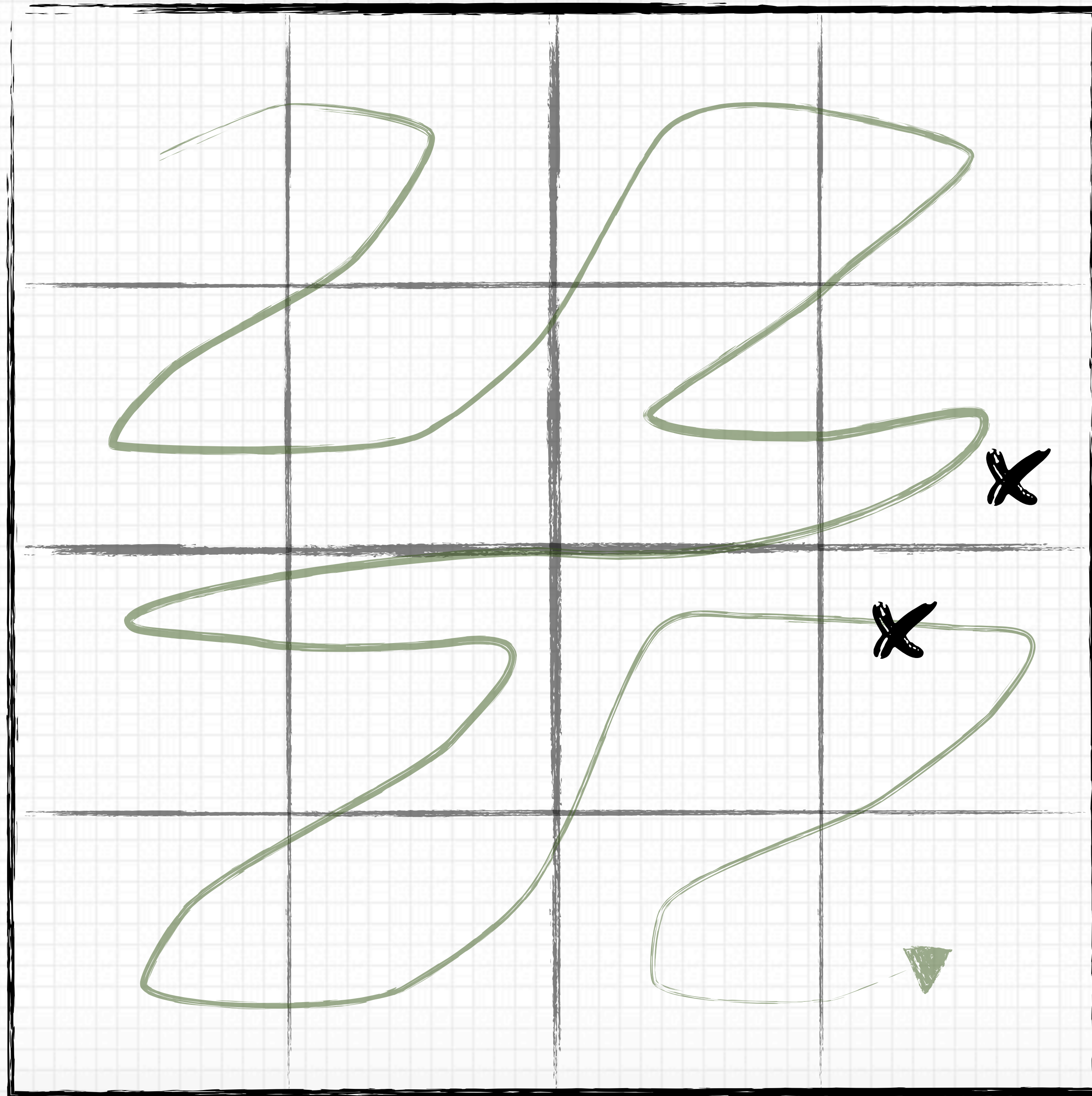
BOUNDING BOX
E.G., MULTIDIMENSIONAL RANGE



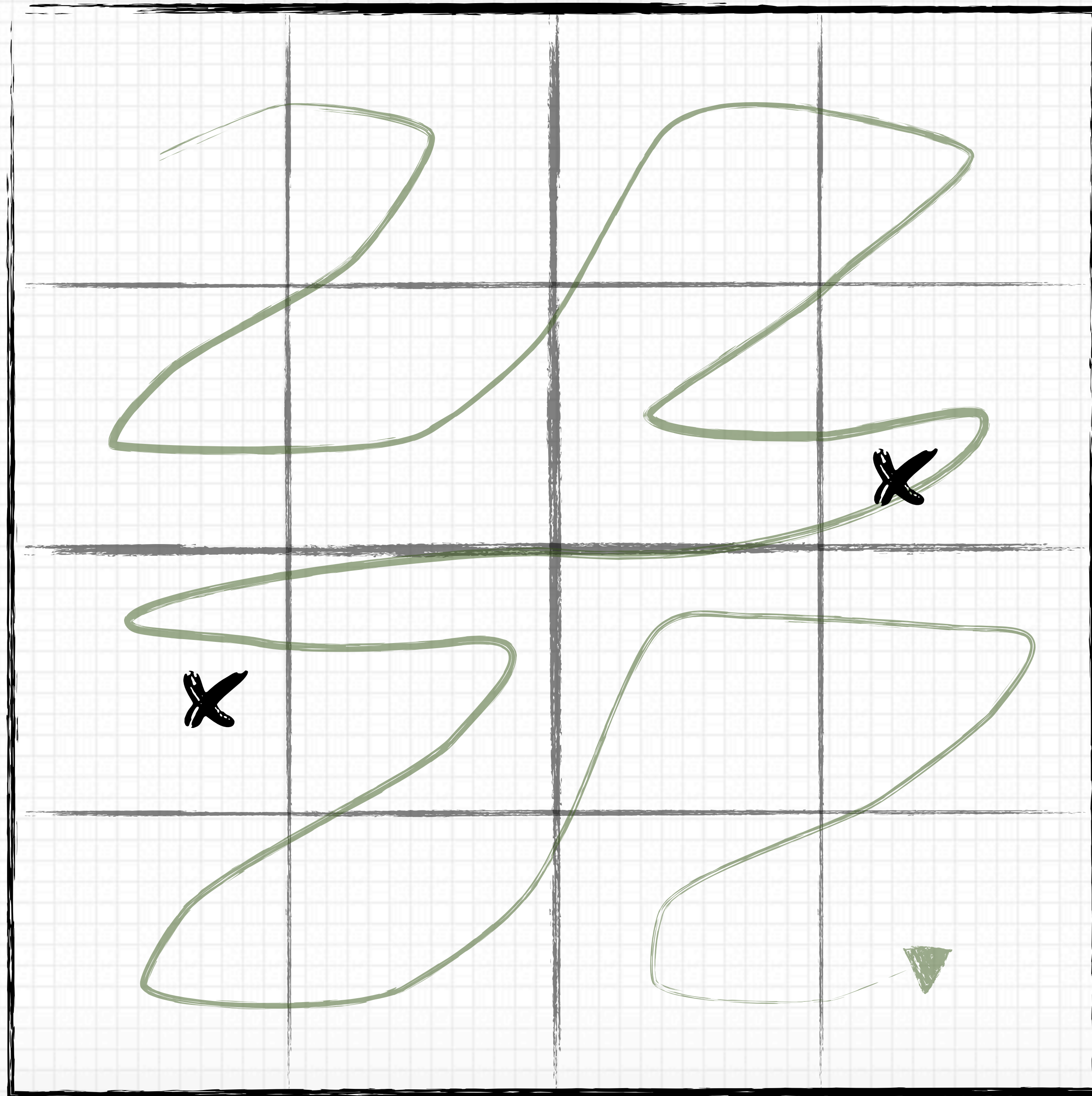
Z-CURVE LOCALITY



Z-CURVE LOCALITY



Z-CURVE LOCALITY



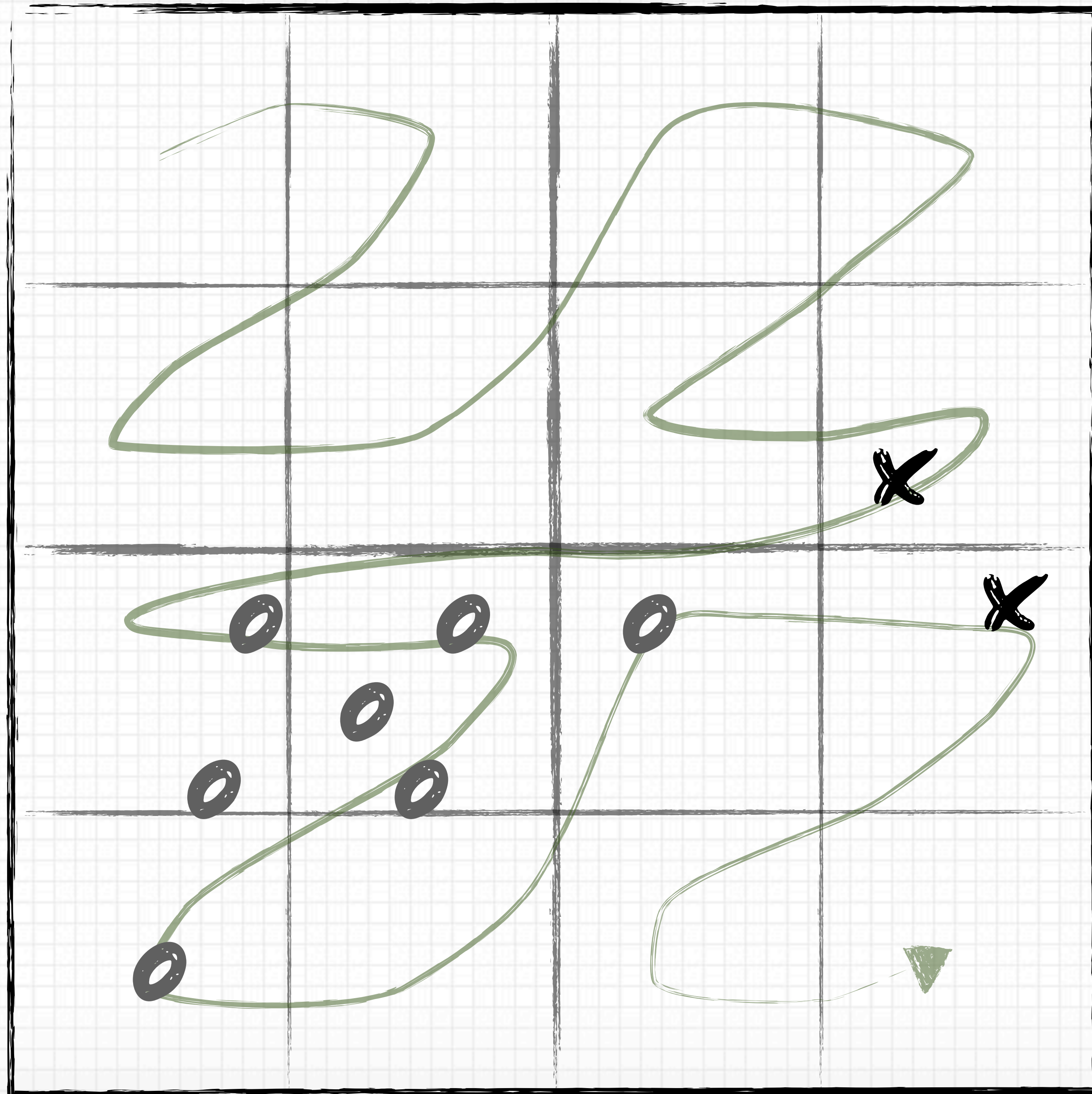
BUT... TURNS OUT SPATIAL DATA IS STILL MULTI-DIMENSIONAL

DIMENSIONALITY REDUCTION ISN'T PERFECT

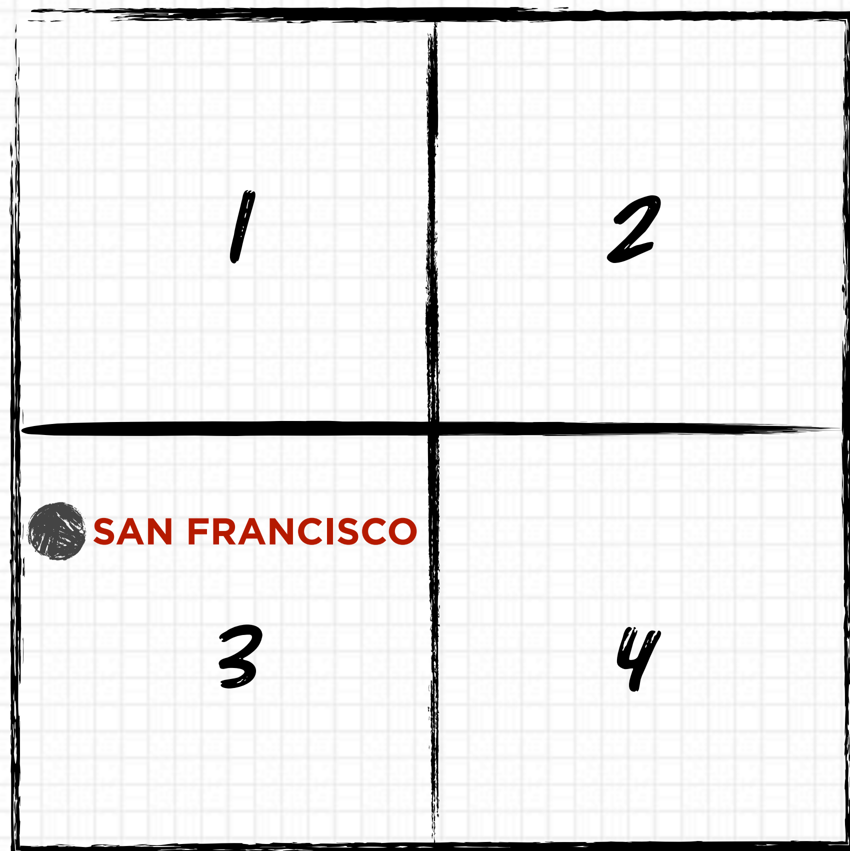
- Sometimes things that are far apart appear close together, and vice versa
- As a result, the client needs to be smart enough to do
  - Pre-processing to compose multiple queries
  - Post-processing to filter and merge results
- This is a more-or-less workable situation for range queries, but breaks down badly for k-nearest-neighbor or proximity queries

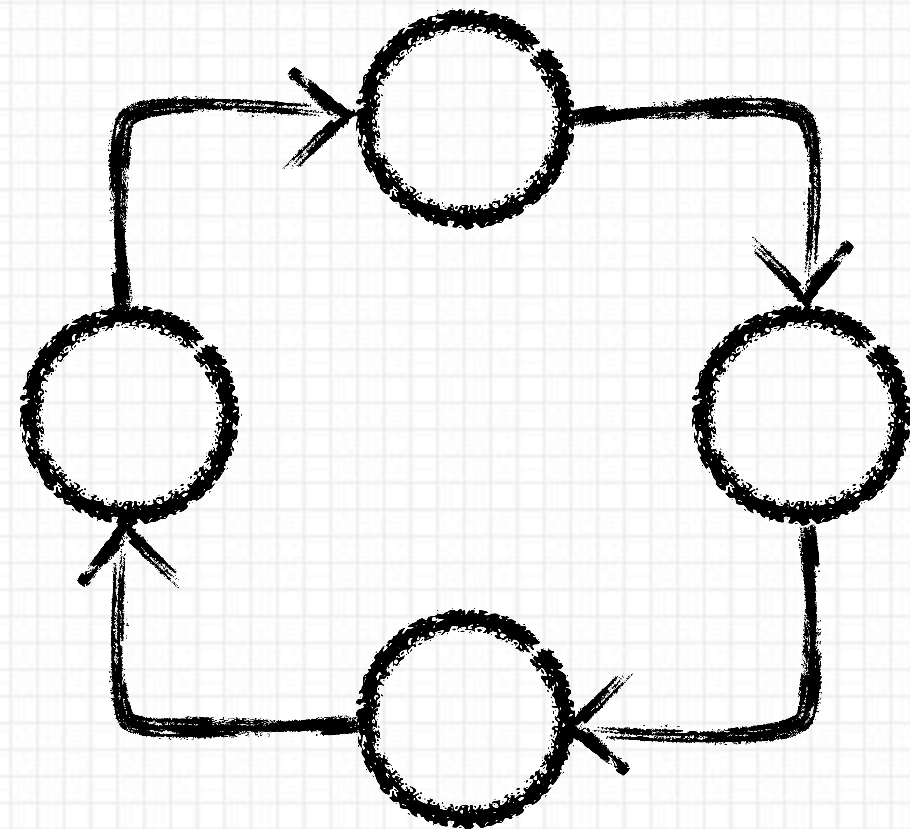
AND IT GETS WORSE WITH ADDITIONAL DIMENSIONS

Z-CURVE LOCALITY

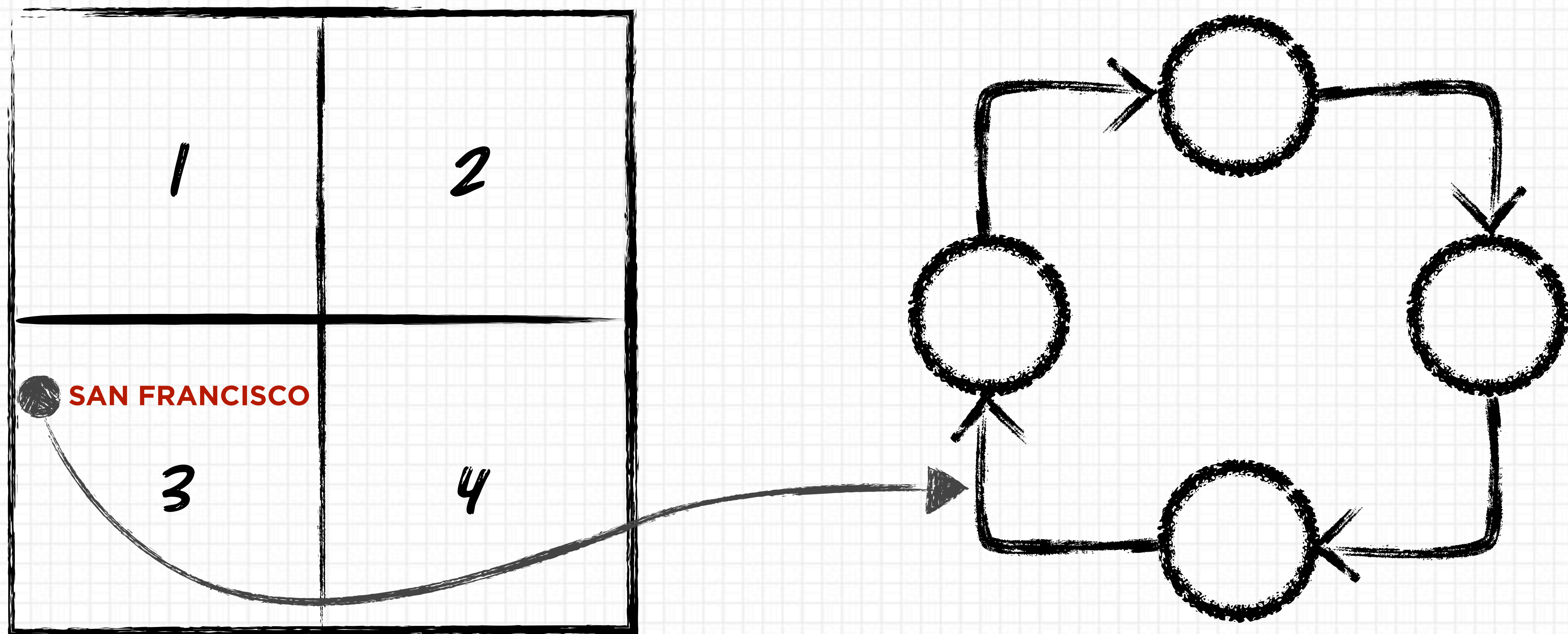


WORSE: TOO MUCH LOCALITY

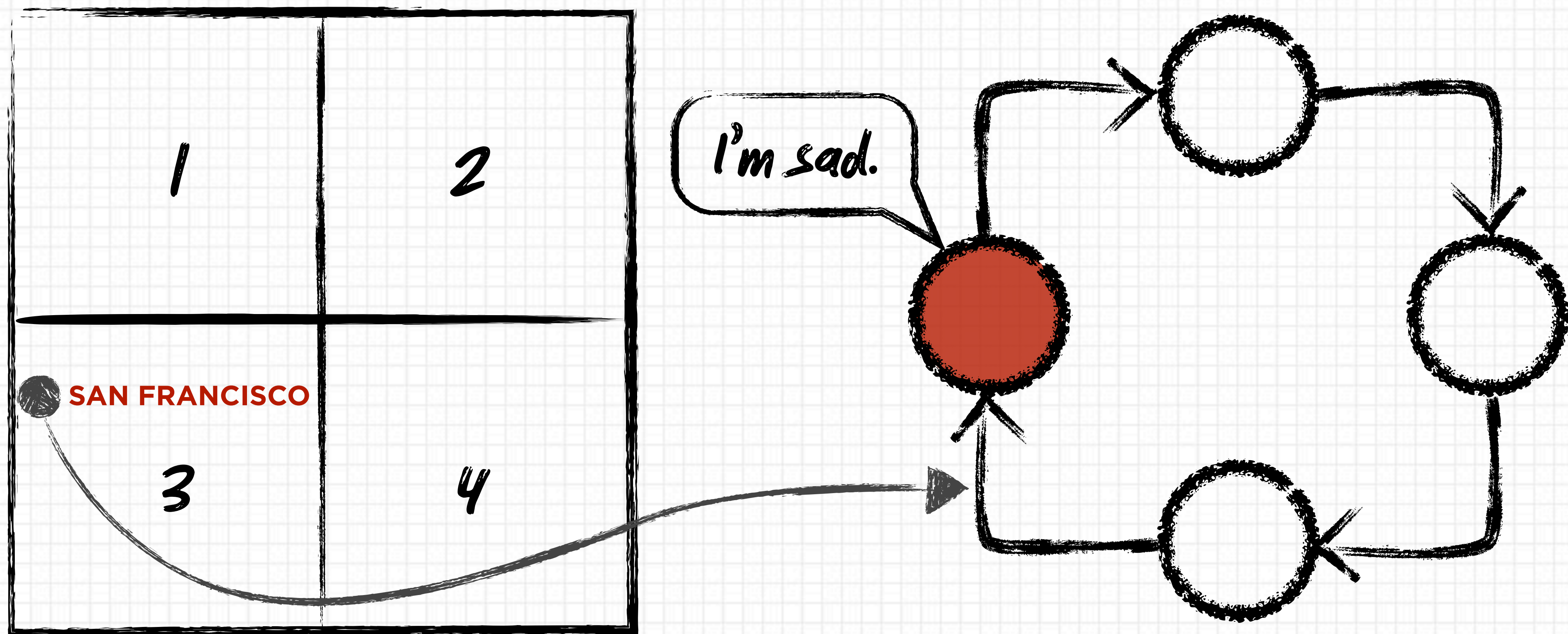




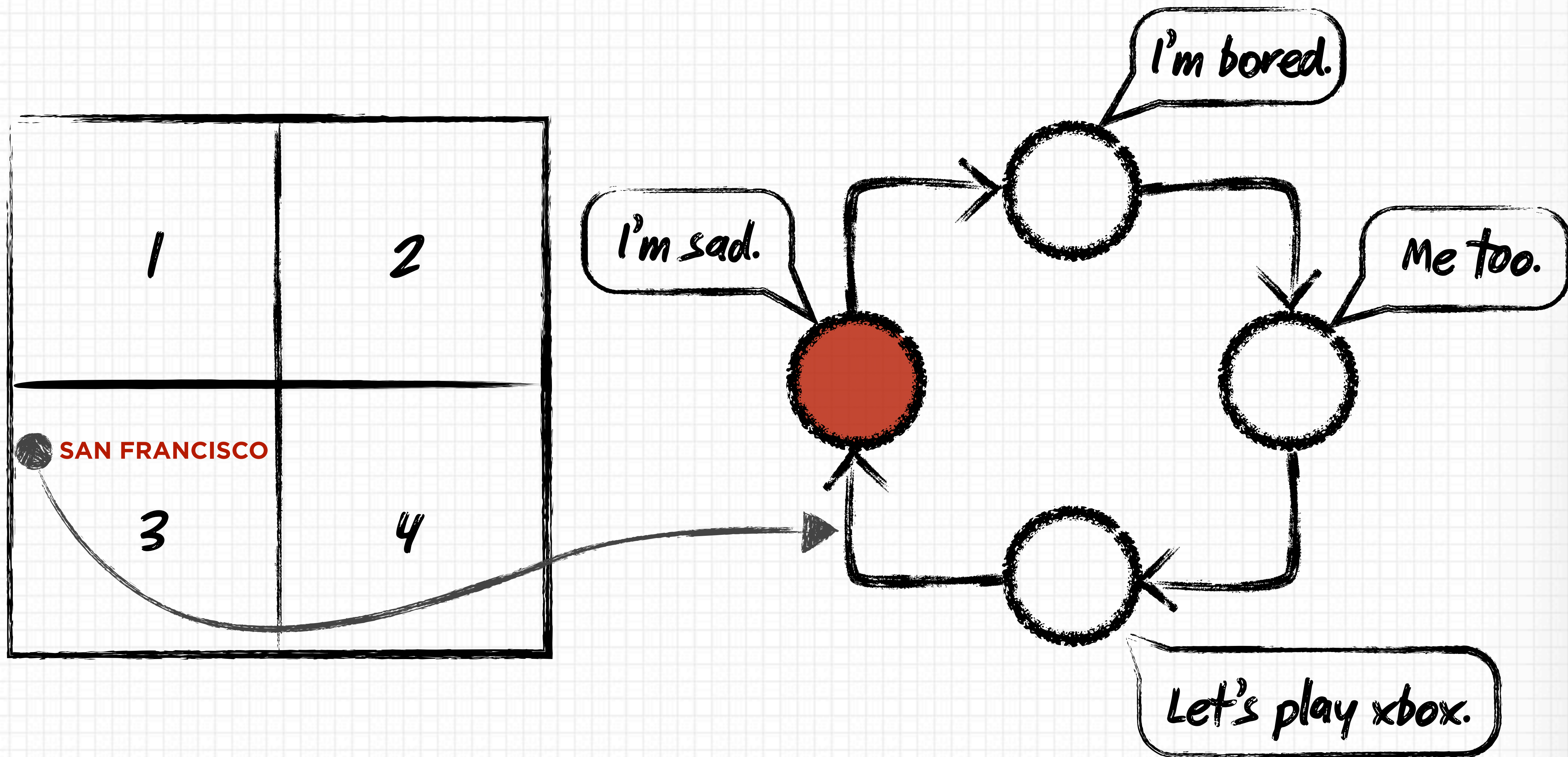
WORSE: TOO MUCH LOCALITY



WORSE: TOO MUCH LOCALITY



WORSE: TOO MUCH LOCALITY



A TURNING POINT

HELLO, DRAWING BOARD

AN EXTREMELY BRIEF SURVEY OF DISTRIBUTED PEER-TO-PEER INDEXING

- An **overlay-dependent** index works directly with nodes of the peer-to-peer network, defining its own overlay
- An **over-DHT** index overlays a more sophisticated data structure on top of a peer-to-peer distributed hash table

BOTH WAYS WORK

- Many of the concepts are actually isomorphic with linear factors differentiating them… it's like building a set out of a dictionary
  - In fact, it's a lot like building an ordered set out of a dictionary

HOW 'BOUT AN OVERLAY-DEPENDENT GRID WITH POSTGIS'?

MIGHT WORK, BUT

- The relational transaction management system (which we'd want to change) and access methods (which we'd have to change) are **tightly coupled** (necessarily?) to other parts of the system
  - Massive re-write would be necessary, most benefit gone
  - This isn't a problem that's specific to PostGIS, but is true of traditional relational databases in general
- Could work at a higher level and treat PostGIS as a black box
  - Now we're back to implementing a peer-to-peer network with failure recovery, fault detection, etc... and Cassandra already had all that.
    - It's probably clear by now that I think these problems are more difficult than actually storing structured data on disk

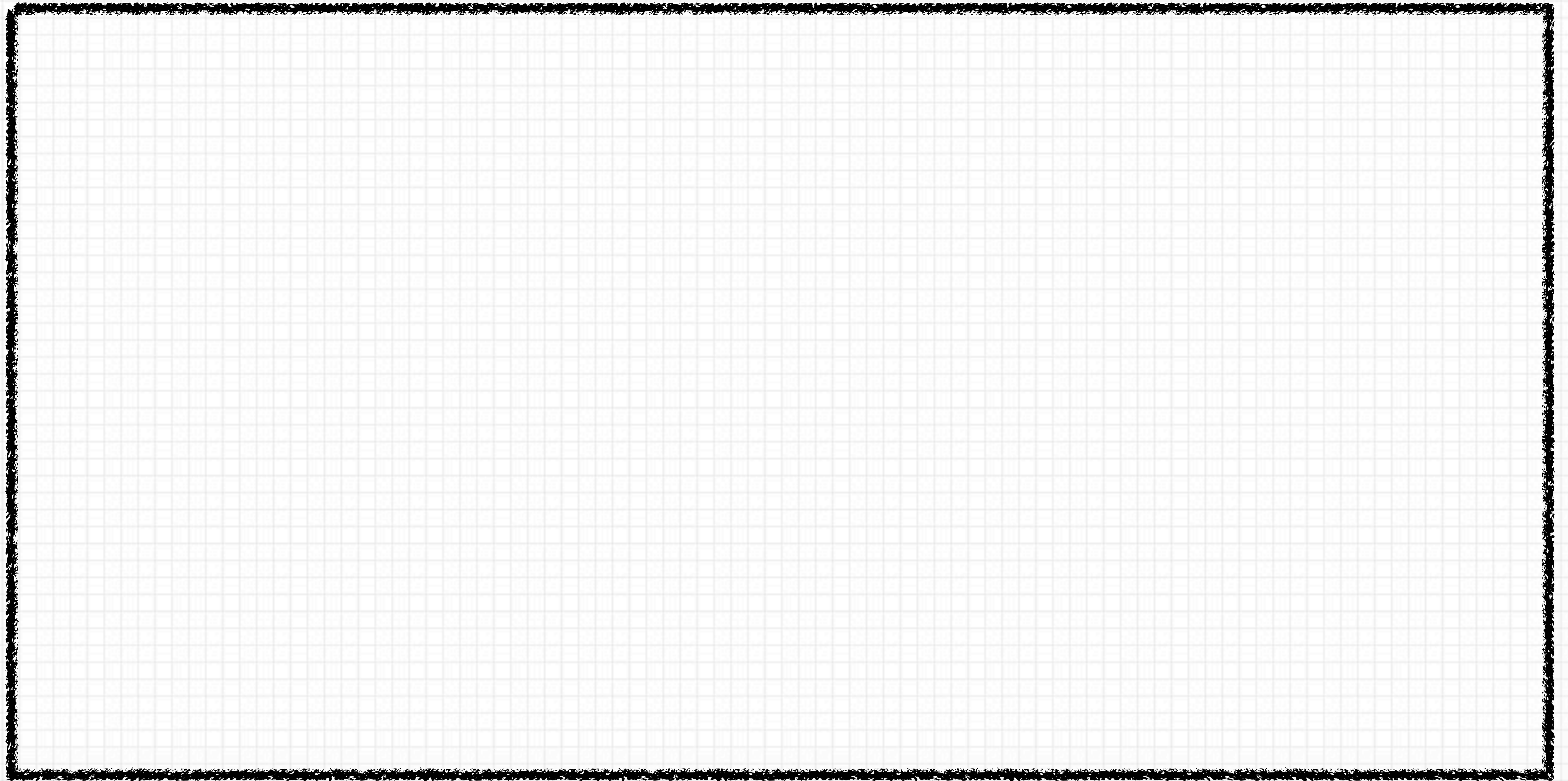
DISTRIBUTED INDEXES OVER-DHT PRIOR ART

LOTS OF ACADEMIC WORK ON THIS TOPIC

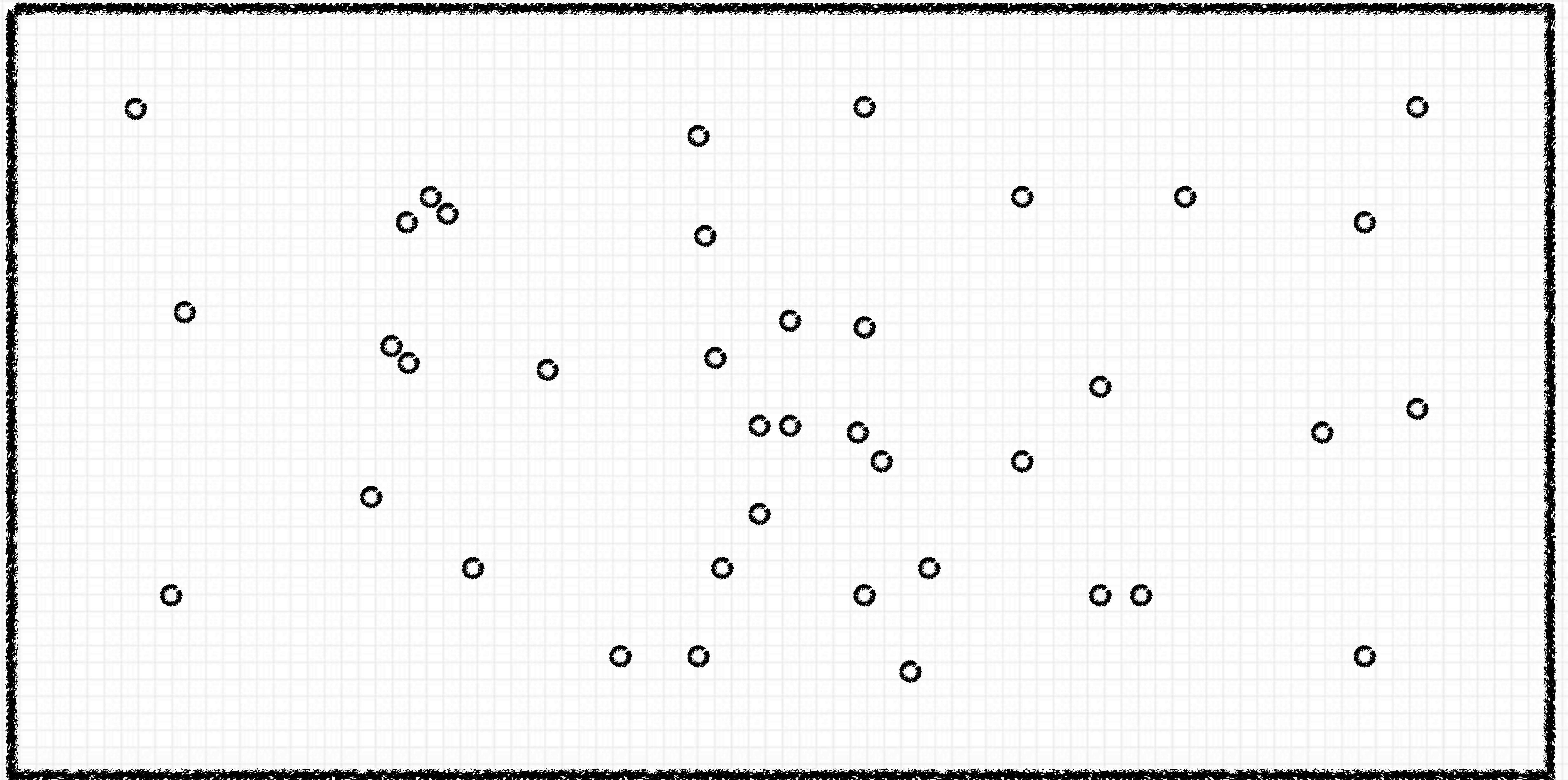
- But academia is obsessed with provable, deterministic, asymptotically optimal algorithms
- And we only need something that is probably fast enough most of the time (for some value of “probably” and “most of the time”)
  - And if the probably good enough algorithm is, you know... tractable... one might even consider it qualitatively better!

LET'S TAKE A STEP BACK

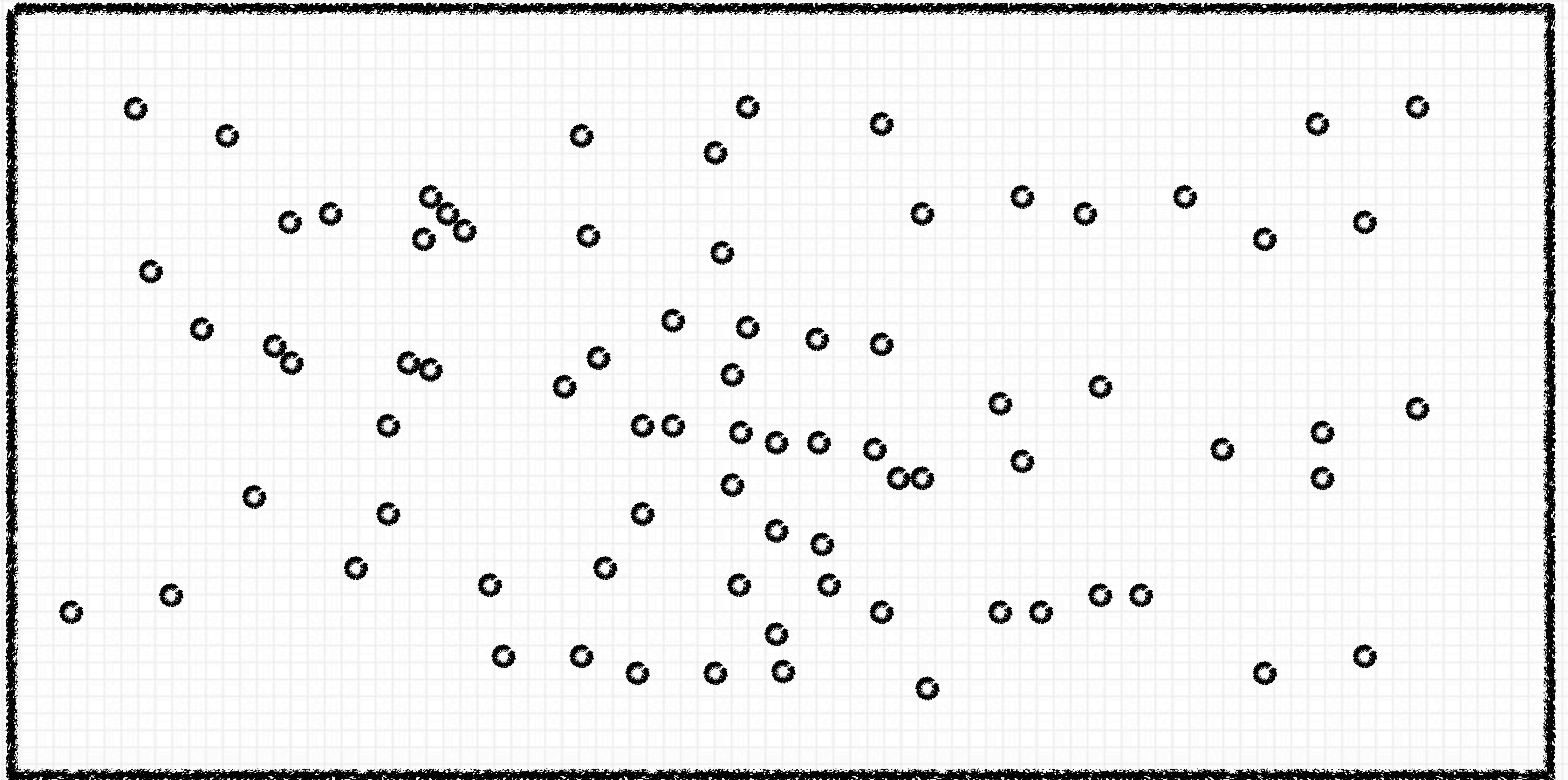
EARTH



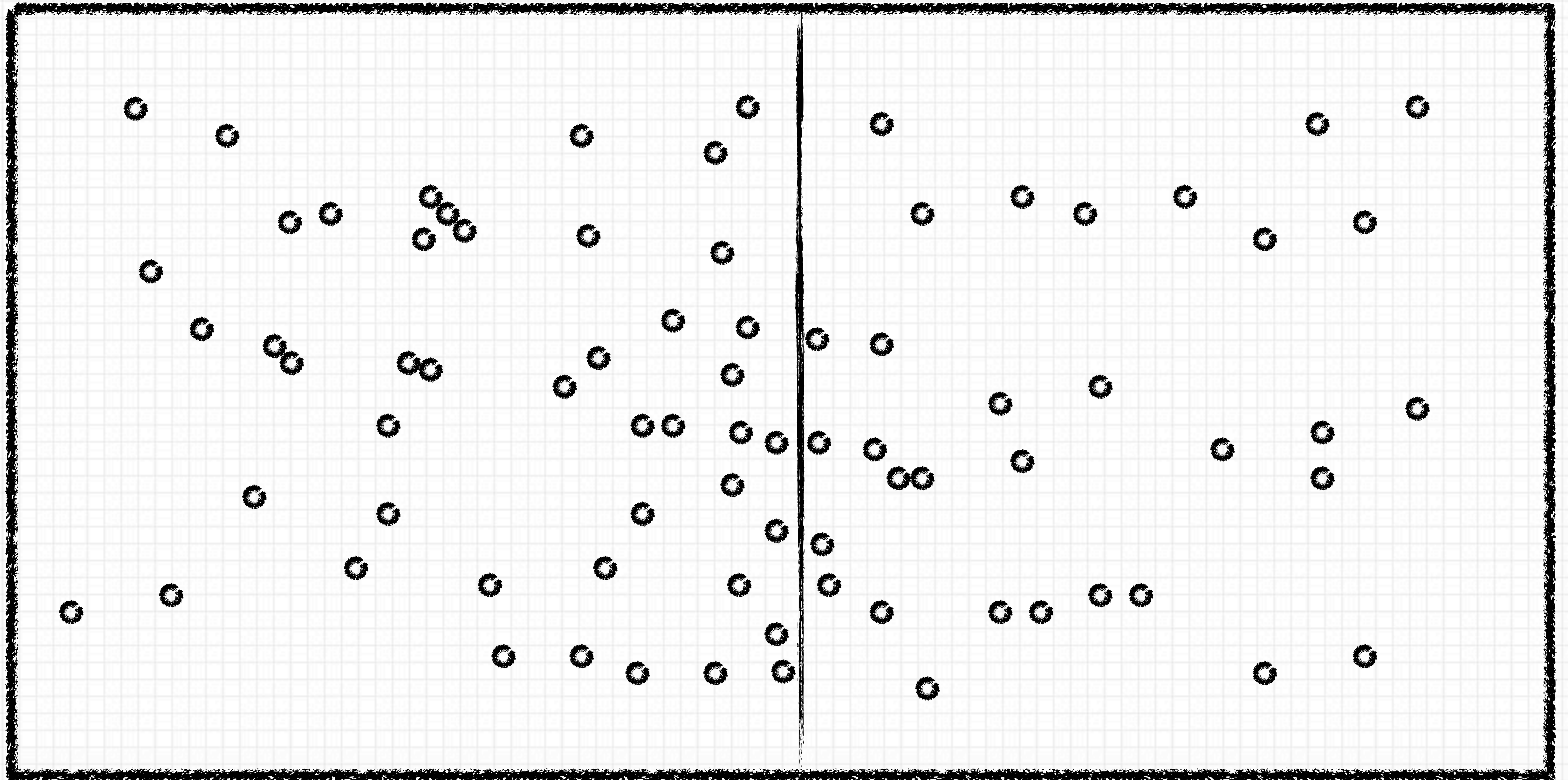
EARTH



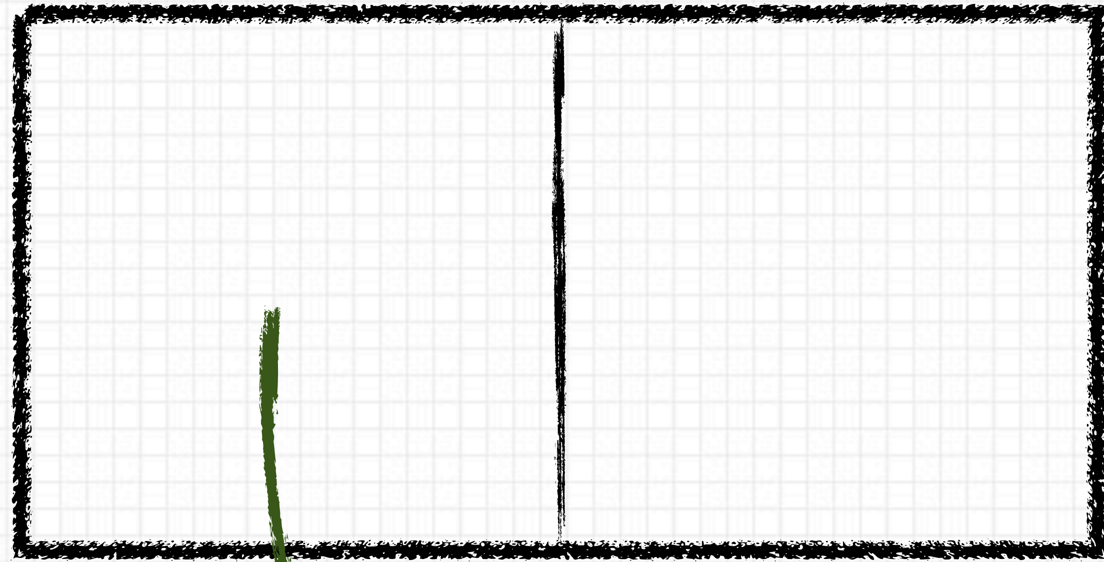
EARTH

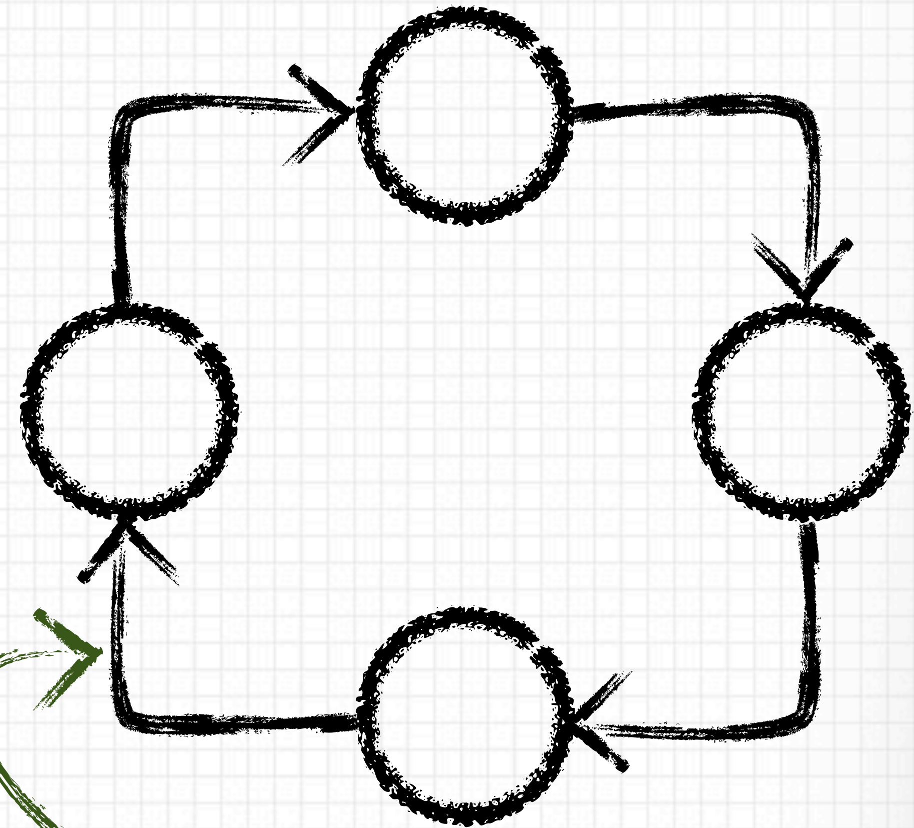


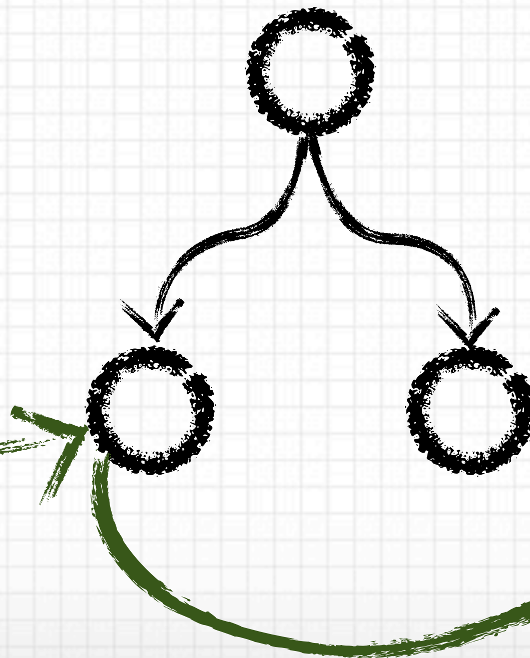
EARTH

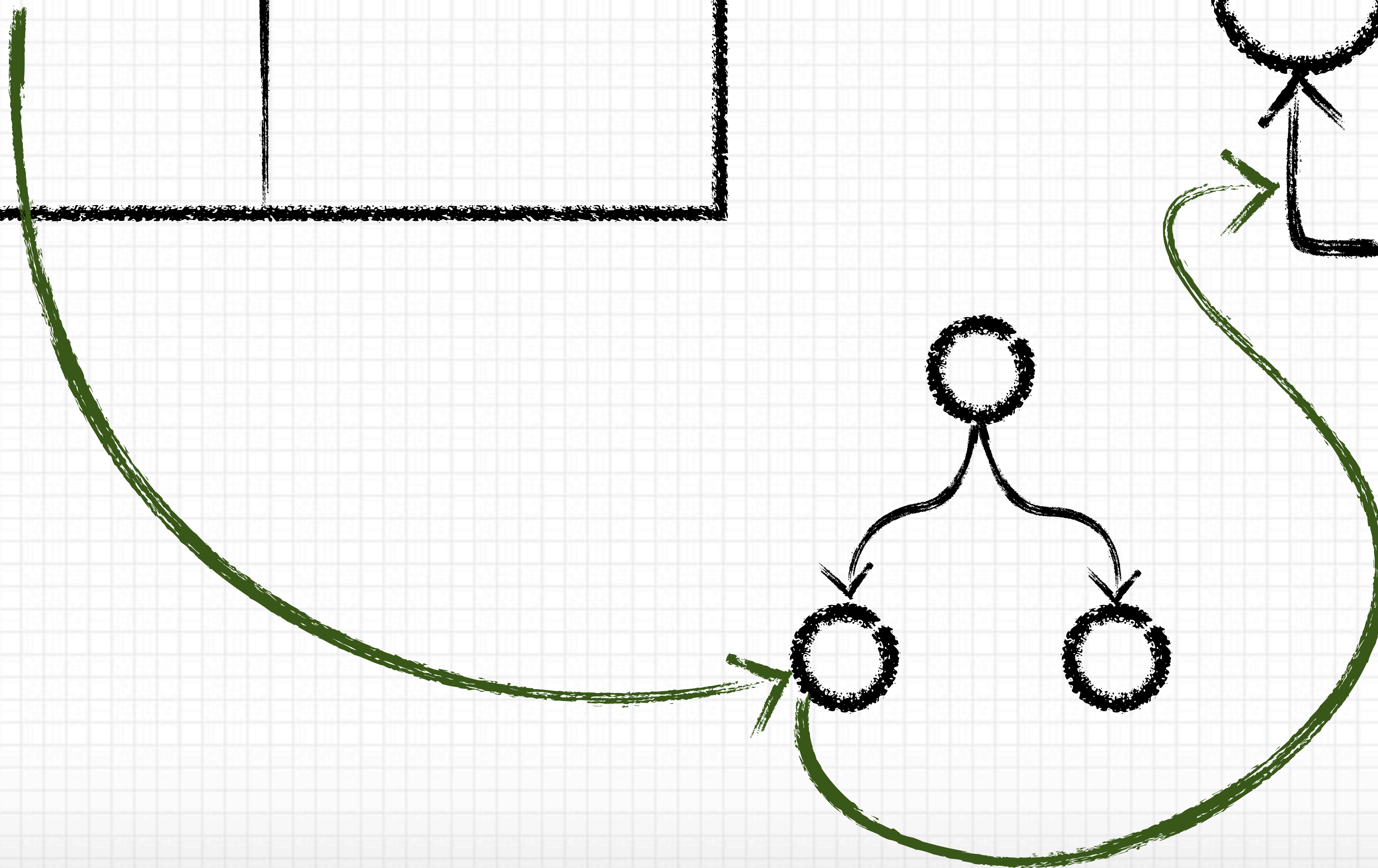


EARTH, TREE, RING









DATA MODEL

EACH NODE HAS

- **Data** items that are being indexed: structured byte arrays containing indexed attributes and identifiers
  - In order to simplify caching and traversal logic we decided to store data in **leaf nodes only**
- **Metadata** about its state: whether it's an internal or leaf node, pointers to its children, and perhaps additional statistical information

SPLITTING
IT'S JUST A CONCURRENT TREE

SPLITTING SHOULDN'T LOCK THE TREE FOR READS OR WRITES AND FAILURES CAN'T CAUSE CORRUPTION

- Splits are optimistic, idempotent, and fail-forward
- Instead of locking, writes are replicated to the splitting node and the relevant child[ren] while a split operation is taking place
  - Cleanup occurs after the split is completed and all interested nodes are aware that the split has occurred
  - Cassandra writes are idempotent, so splits are too - if a split fails, it is simply be retried

SPLIT SIZE: A TUNABLE KNOB FOR BALANCING LOCALITY AND DISTRIBUTEDNESS

THE ROOT IT'S SO HOT

HARD PROBLEM

- For a tree to be useful, it has to be traversed
  - Typically, tree traversal starts at the root
  - Further, the only discoverable node in our tree is the root, which limited our options
  - But traversing through the root means reading the root, and reading the root for every traversal (read *and* write) was unacceptable
- Again, lots of academic solutions - most promising was a skip graph, but that required `O(n log(n))` data - also unacceptable
- Some proposals suggest using a minimum tree depth, but that's a bandaid on a bullet wound: you just get multiple hot-spots at your minimum depth

THE ROOT IT'S SO HOT

STUPID SIMPLE SOLUTION

- Keep an LRU cache of nodes that have been traversed
- Start traversals at the most selective relevant node
- If that node doesn't satisfy you, traverse **up** the tree
- Along with your result set, return a list of nodes that were traversed so the caller can add them to its cache

PERFORMANCE CHARACTERISTICS

- Best case on the happy path (everything cached) has zero read overhead
- Worst case, with nothing cached, `O(log(n))` read overhead

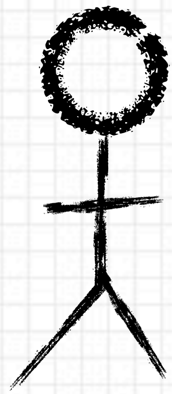
RE-BALANCING IS MOSTLY UNNECESSARY!

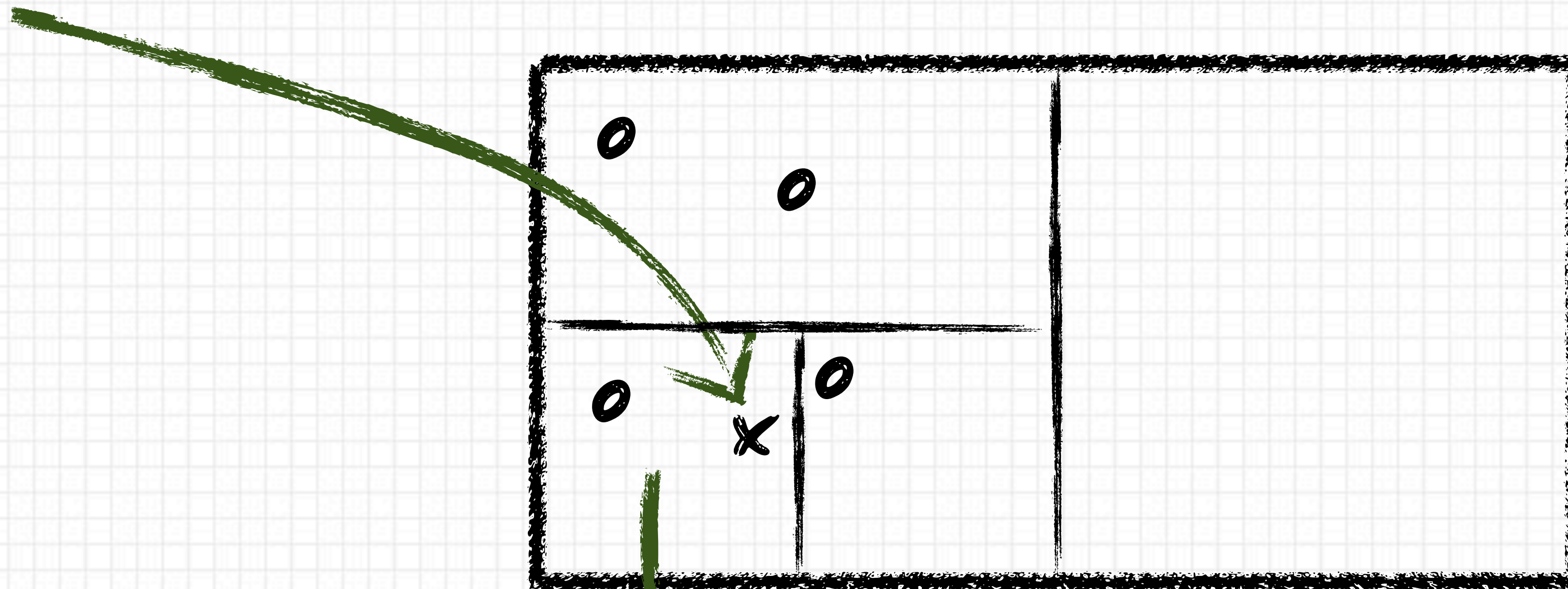
TRAVERSALS

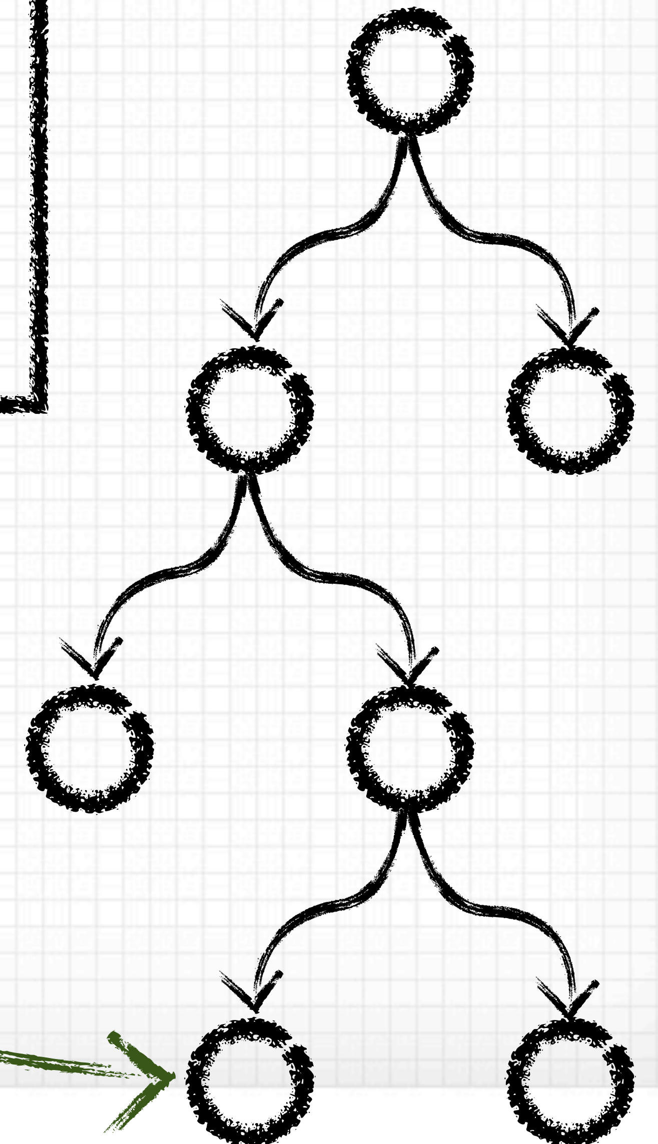
THEY GO BOTH WAYS

- If traversals can start anywhere in the tree, they need to be careful to cover every node that is relevant to it, which may mean traversing down *and* up the tree

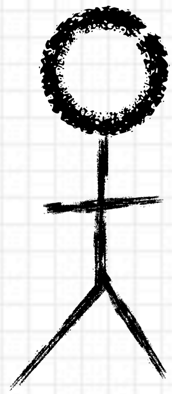
TRAVERSAL
NEAREST NEIGHBOR

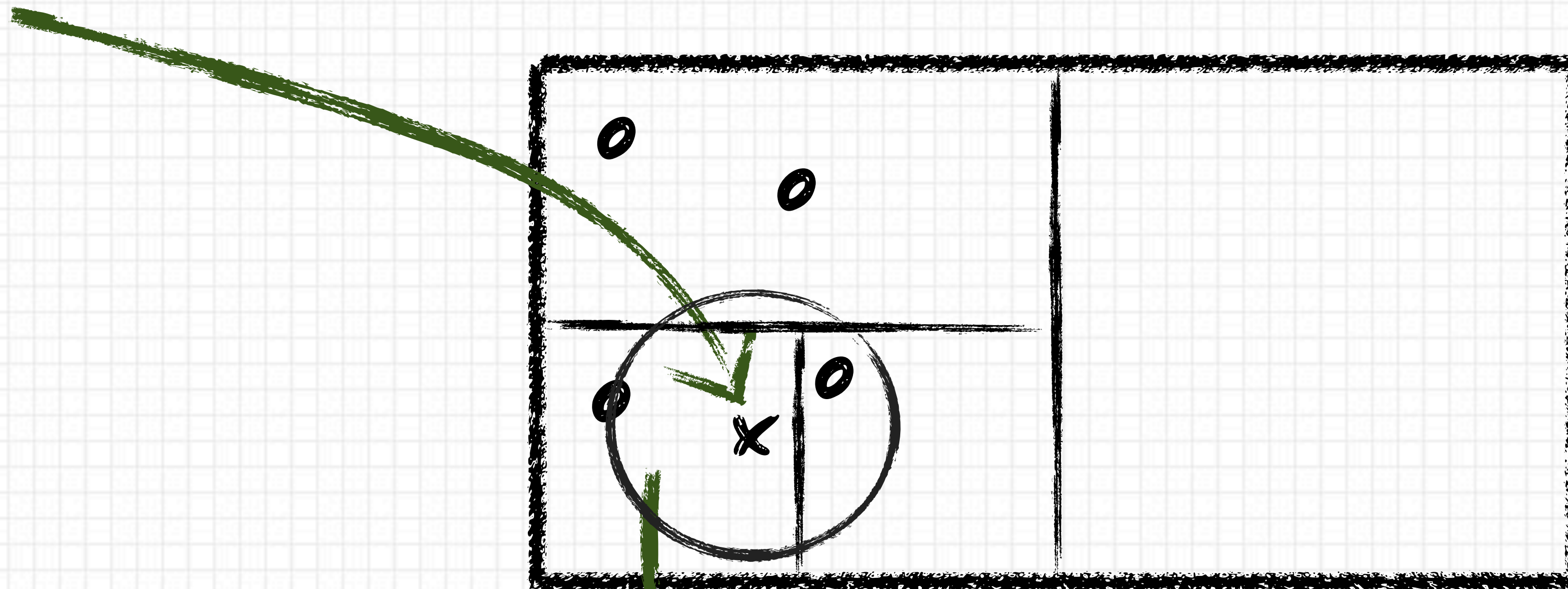


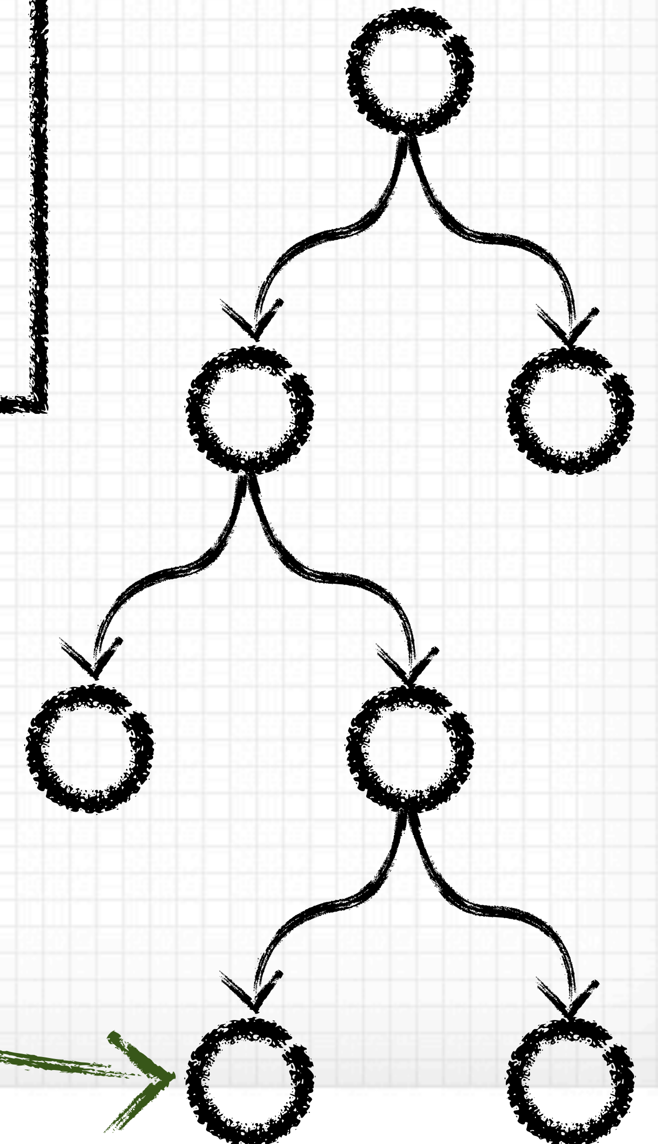




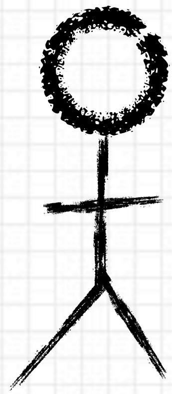
TRAVERSAL
NEAREST NEIGHBOR

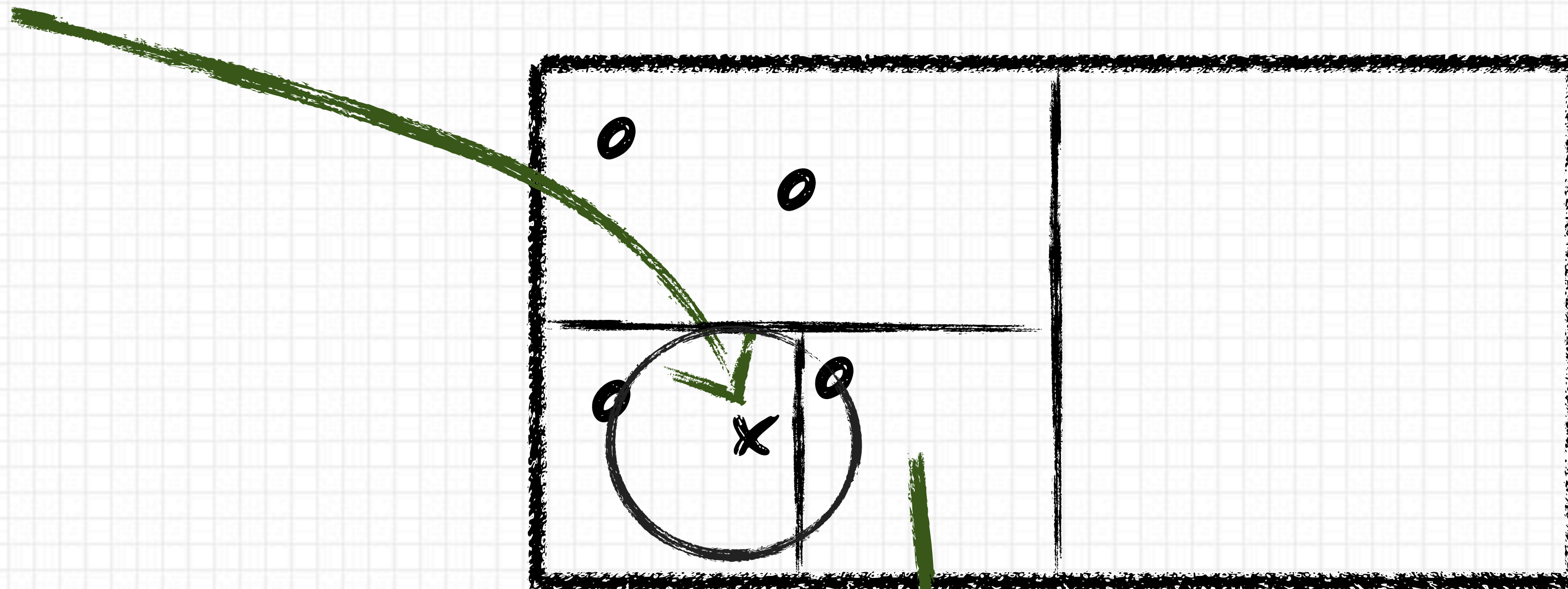


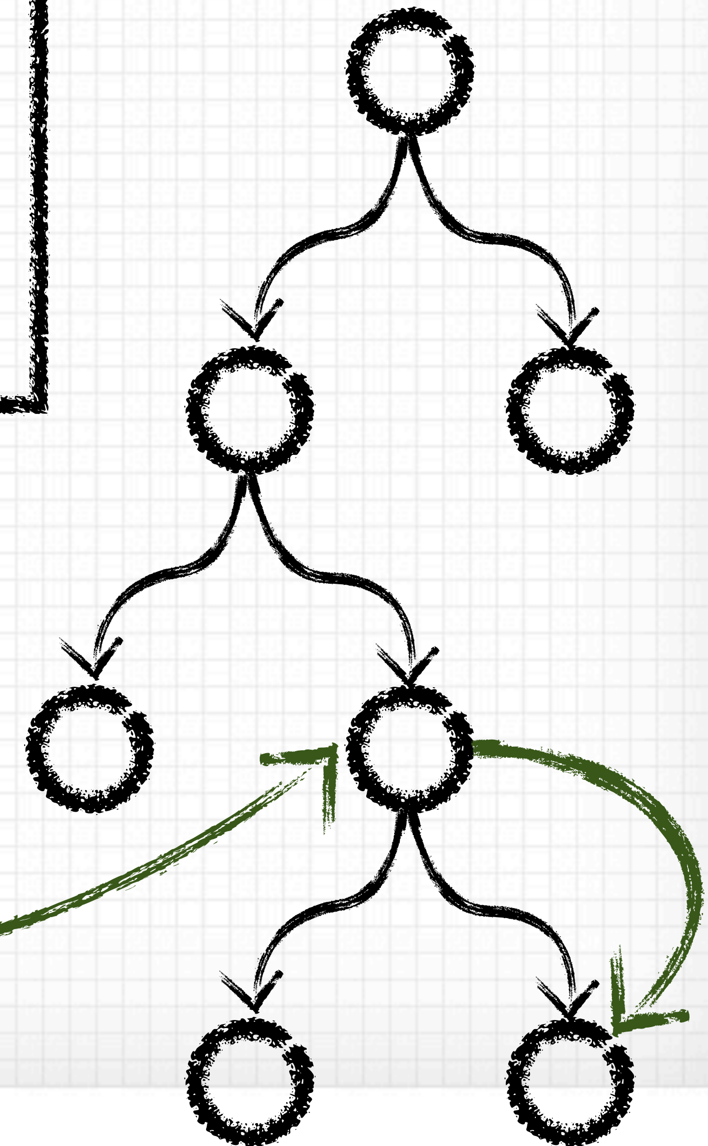




TRAVERSAL NEAREST NEIGHBOR

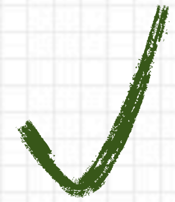


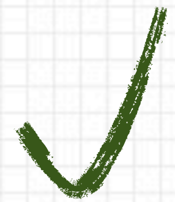


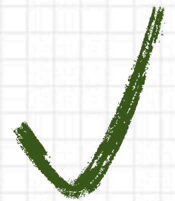


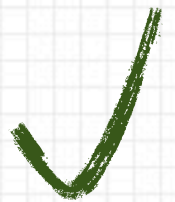
CLIMAX

DISTRIBUTED TREE SUPPORTED QUERIES


EXACT MATCH


RANGE


PROXIMITY


SOMETHING ELSE I HAVEN'T EVEN HEARD OF

DISTRIBUTED TREE SUPPORTED QUERIES

- ✓ EXACT MATCH
- ✓ RANGE
- ✓ PROXIMITY
- ✓ SOMETHING ELSE I HAVEN'T EVEN HEARD OF

MULTIPLE DIMENSIONS!

DESIRABLE CHARACTERISTICS

- ✓ HIGHLY AVAILABLE
- ✓ FAULT TOLERANT
- ✓ DECENTRALIZED
- ✓ HORIZONTALLY SCALABLE
- ✓ OPERATIONALLY SIMPLE

THE END?

FUTURE IMPROVEMENTS

GENERALIZE TO GRAPHS

- A tree is a degenerate graph
- I believe this approach would work for any graph for which you can do an online computation of strongly connected subgraphs and the connections are relatively stable
- Would probably need a more sophisticated bridging model
  - Something like Valiant's Bulk-Synchronous Parallel model (used in Google Pregel)

FUTURE IMPROVEMENTS

AGGREGATIONS UP THE TREE

- Statistics about the data in the leaf node can be bubbled up the tree to provide efficient parameter estimates at various levels of granularity
- When you really embrace eventual consistency and best-effort style systems a lot of intractable problems become tractable in generic ways

QUESTIONS?

MIKE MALONE

INFRASTRUCTURE ENGINEER

mike@simplegeo.com
@mjmalone

SimpleGeo™

GOTO 2010