An overview of

Neo4j Internals

Tobias Lindaaker

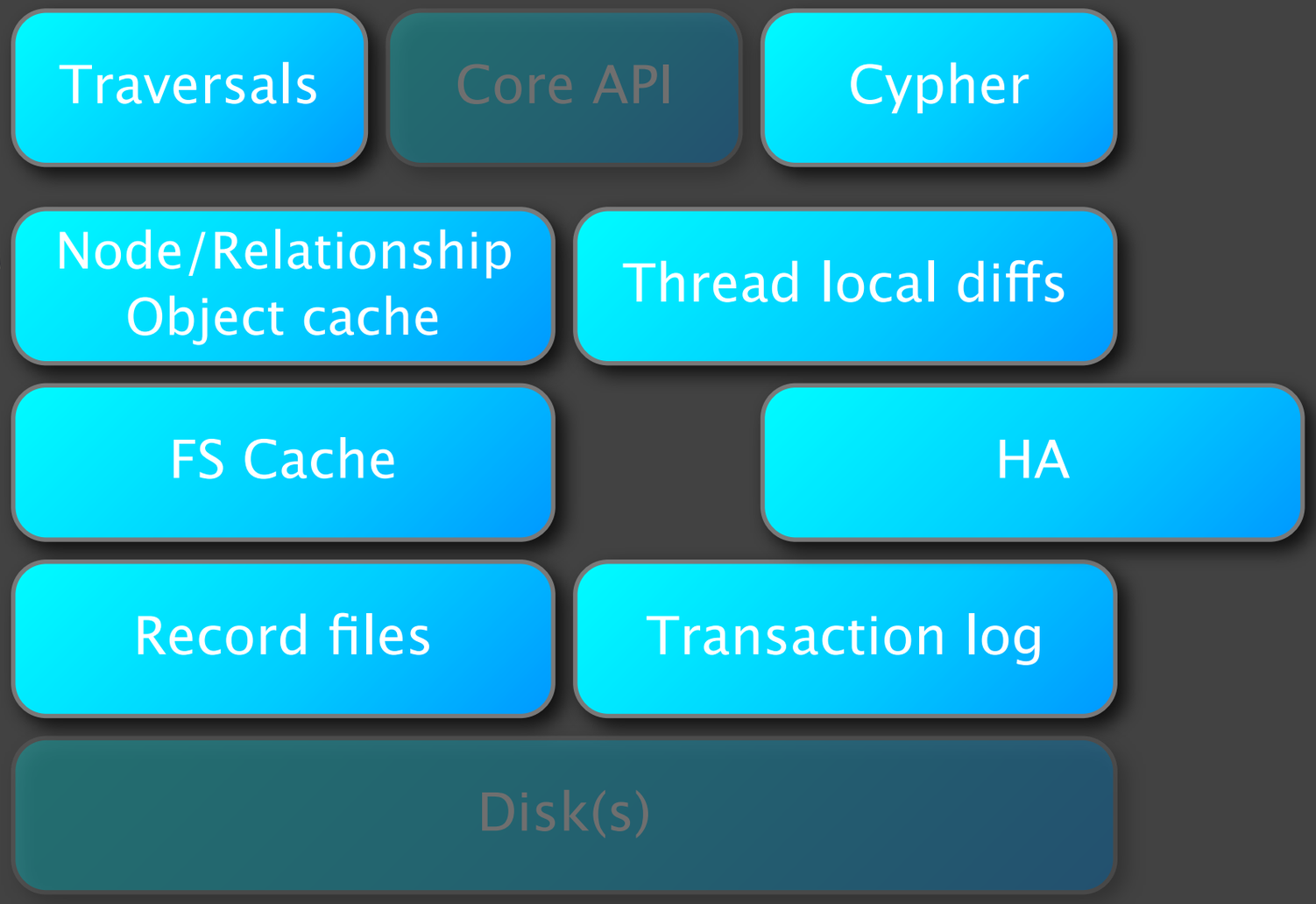
Hacker @ Neo Technology

tobias@neotechnology.com
twitter: @thobe, #neo4j (@neo4j)
web: neo4j.org neotechnology.com
my web: thobe.org

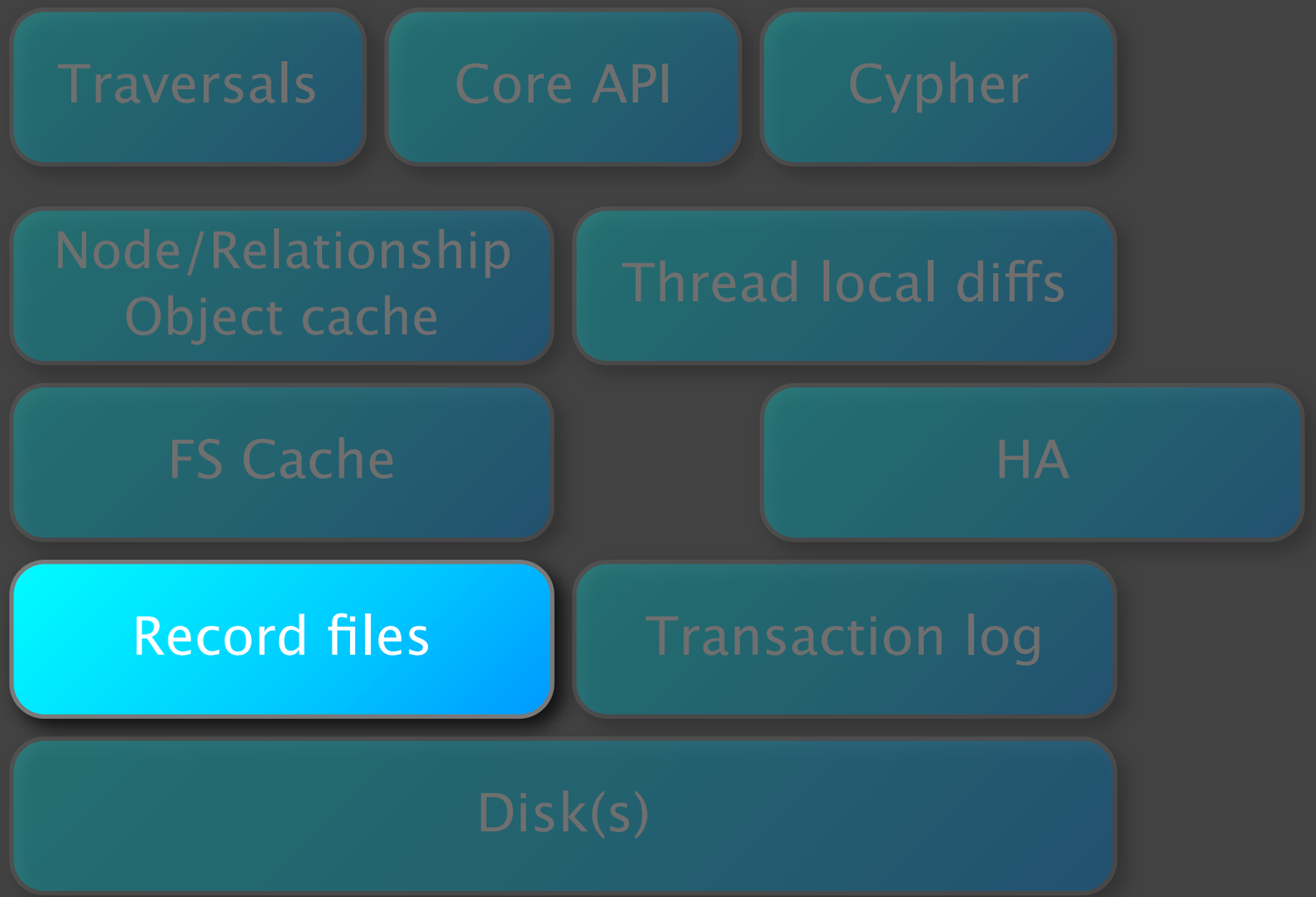
Outline

This is a rough structure of how the pieces of Neo4j fit together.

This talk will not cover how disks/fs works, we just assume it does.
Nor will it cover the "Core API", you are assumed to know it.



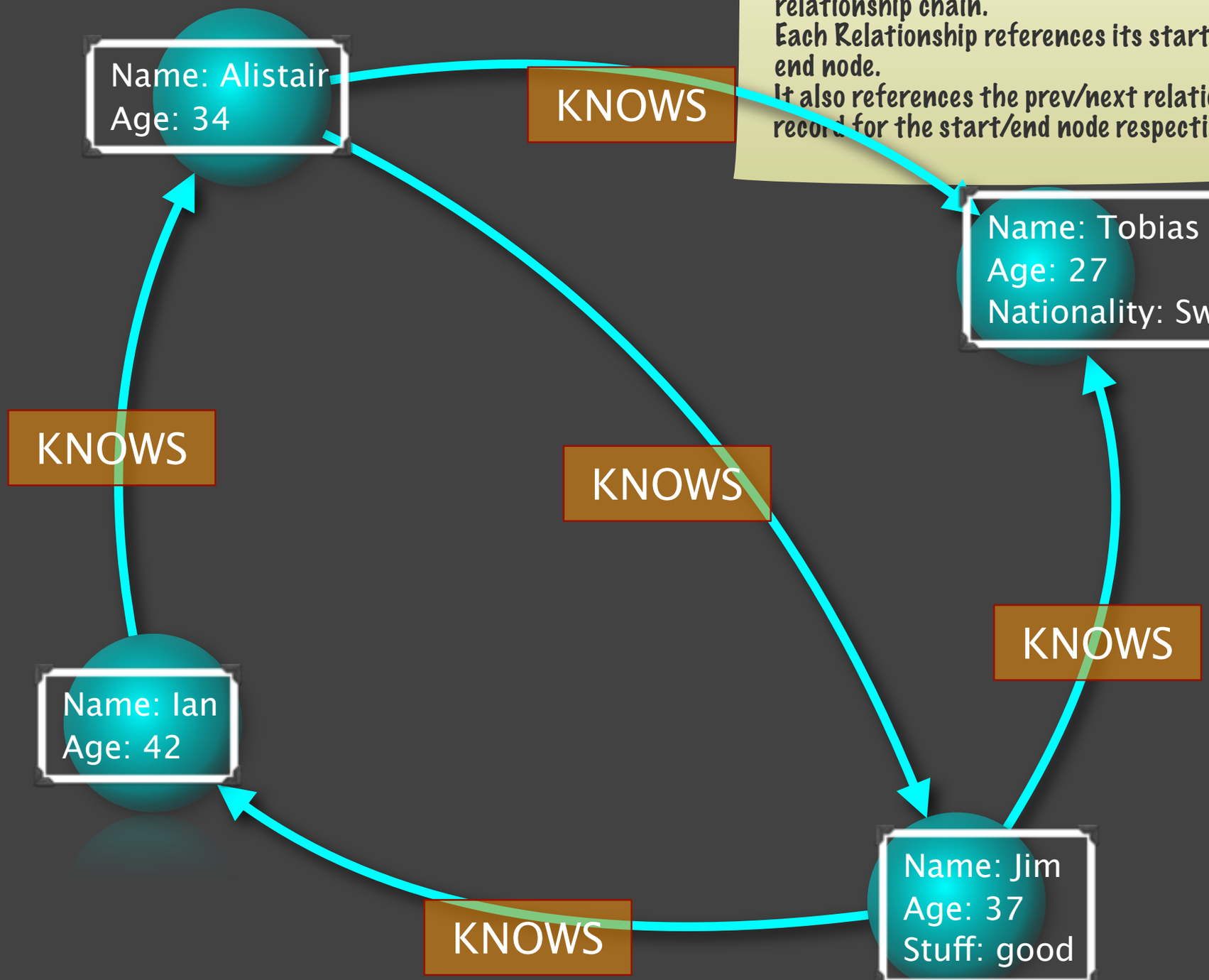
Outline



Let's start at the bottom: the on disk storage file layout.

Your graph on disk

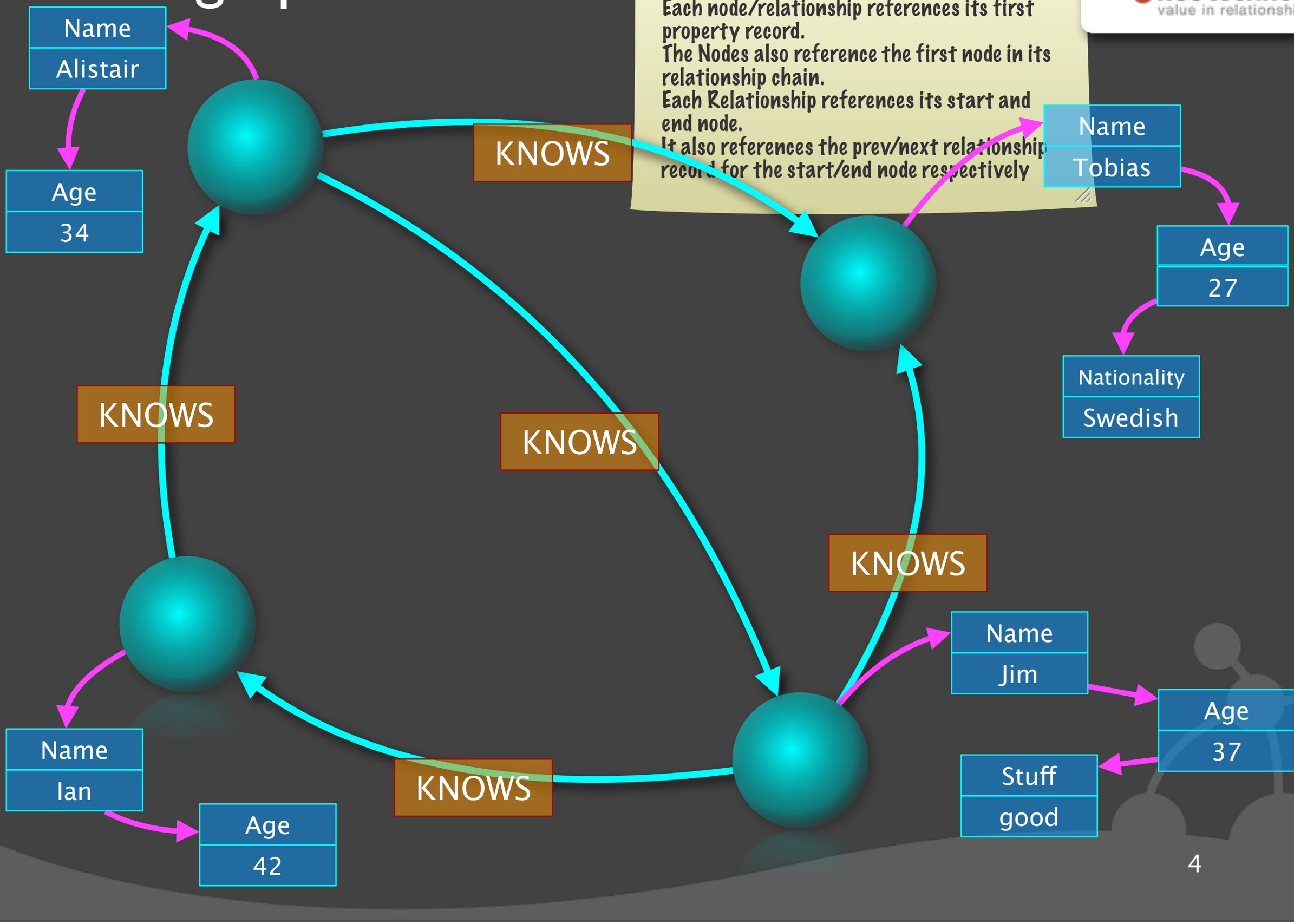
Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively



Your graph on disk

Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively

neo technology
value in relationships



Your graph on disk

Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively

neo technology
value in relationships

Name
Alistair

Age
34

KNOWS

Name
Tobias

Age
27

Nationality
Swedish

KNOWS

KNOWS

KNOWS

KNOWS

Name
Jim

Age
37

Stuff
good

Name
Ian

Age
42

Your graph on disk

Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively

neo technology
value in relationships

Name
Alistair

Age
34

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

Name
Tobias

Age
27

Nationality
Swedish

SP EP
SN EN
KNOWS

Name
Jim

Age
37

Stuff
good

Name
Ian

Age
42

Your graph on disk

Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively

neo technology
value in relationships

Name
Alistair

Age
34

SP EP
SN EN
KNOWS

Name
Tobias

Age
27

Nationality
Swedish

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

Name
Jim

Age
37

Stuff
good

Name
Ian

Age
42

SP EP
SN EN
KNOWS

Your graph on disk

Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively

neo technology
value in relationships

Name
Alistair

Age
34

SP EP
SN EN
KNOWS

Name
Tobias

Age
27

Nationality
Swedish

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

SP EP
SN EN
KNOWS

Name
Ian

Age
42

Name
Jim

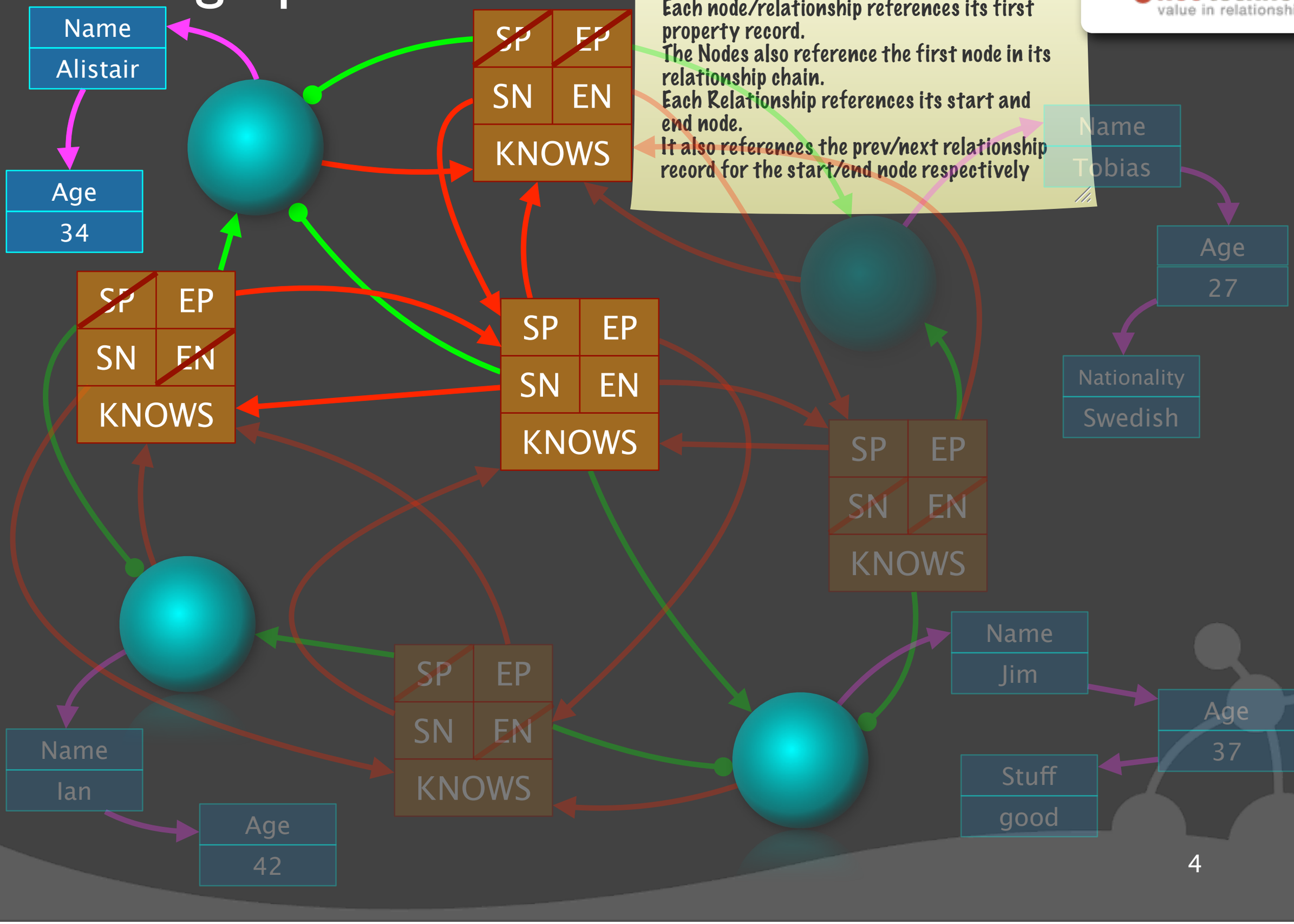
Age
37

Stuff
good

Your graph on disk

Simple sample graph. It all boils down to linked lists of fixed size records on disk.
Properties are stored as a linked list of property records, each holding key+value.
Each node/relationship references its first property record.
The Nodes also reference the first node in its relationship chain.
Each Relationship references its start and end node.
It also references the prev/next relationship record for the start/end node respectively

neo technology
value in relationships



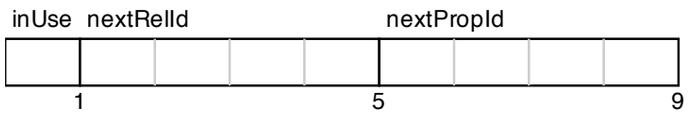
Store files

- Node store
- Relationship store
  - Relationship type store
- Property store
  - Property key store
  - (long) String store
  - (long) Array store

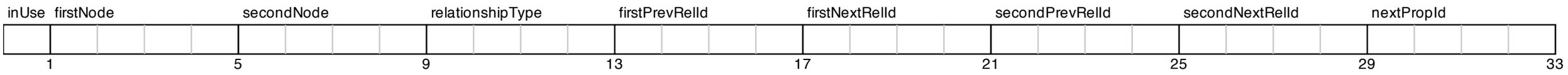
Short string and array values are inlined in the property store, long values are stored in separate store files.

Neo4j Storage Record Layout

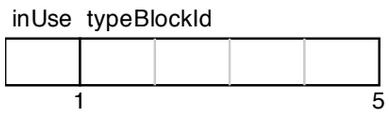
Node (9 bytes)



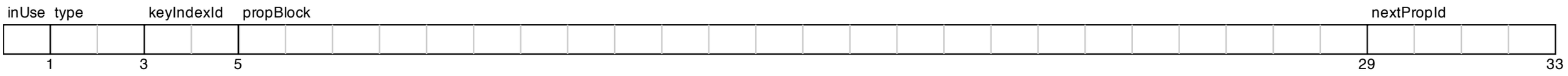
Relationship (33 bytes)



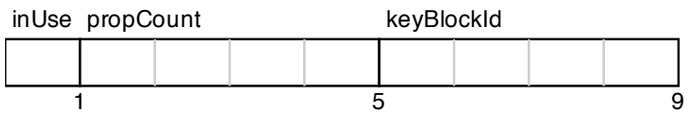
Relationship Type (5 bytes)



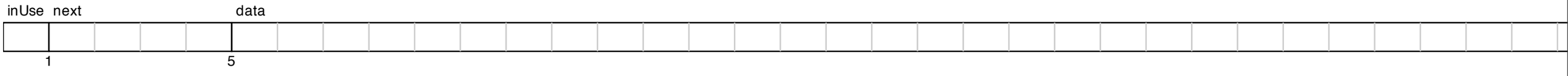
Property (33 bytes)



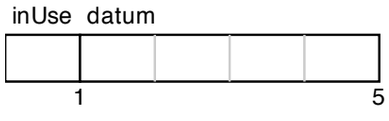
Property Index (9 bytes)



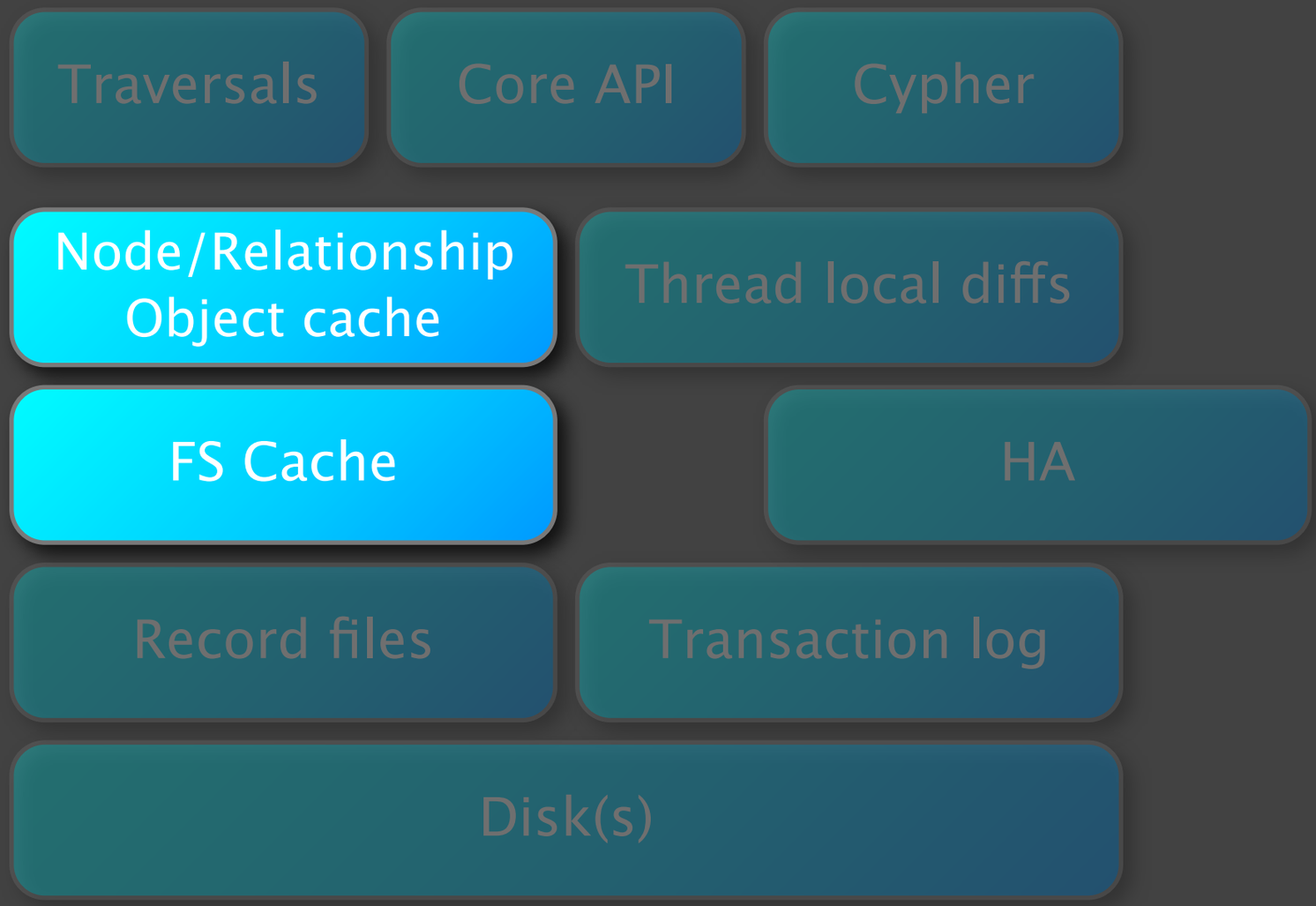
Dynamic Store (125 bytes)



NeoStore (5 bytes)



Outline

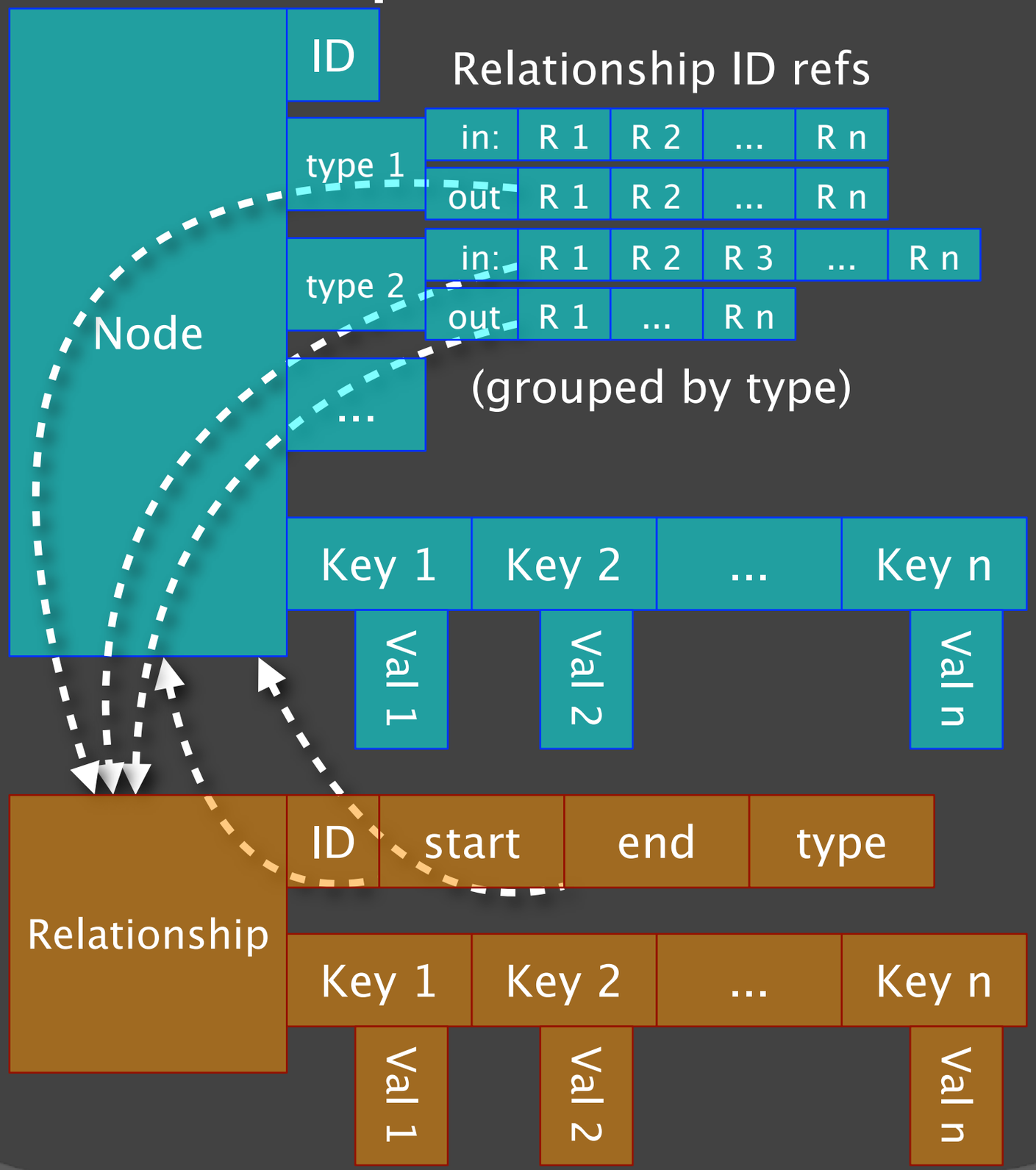


Next: The two levels of cache in Neo4j.
The low level FS Cache for the record files.
And the high level Object cache storing a structure more optimized for traversal.

The caches

- Filesystem cache:
  - Caches regions of the store files
    (divides each store file into equally sized regions)
  - The cache holds a fixed number of regions for each file
  - Regions are evicted based on a LFU-like policy
    (hit count vs. miss count, i.e. hit in non-cached region)
  - Default implementation of regions uses OS mmap
- Node/Relationship cache
  - Cache a version more optimized for traversals

What we put in cache



The structure of the elements in the high level object cache.

On disk most of the information is contained in the relationship records, with the nodes just referencing their first relationship. In the cache this is turned around: the nodes hold references to all its relationships. The relationships are simple, only holding its properties.

The relationships for each node is grouped by RelationshipType to allow fast traversal of a specific type.

All references (dotted arrows) are by ID, and traversals do indirect lookup through the cache.

Outline

So how do traversals work...

Traversals

Core API

Cypher

Node/Relationship Object cache

Thread local diffs

FS Cache

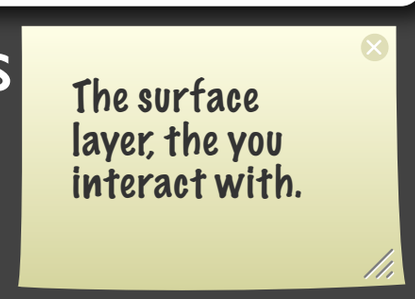
HA

Record files

Transaction log

Disk(s)

Traversals - how do they work?



- ***RelationshipExpander****s*: given (a path to) a node, returns Relationships to continue traversing from that node
- ***Evaluator****s*: given (a path to) a node, returns whether to:
  - Continue traversing on that branch (i.e. *expand*) or not
  - Include (the path to) the node in the result set or not
- Then a projection to `Path`, `Node`, or `Relationship` applied to each Path in the result set.

... but also:

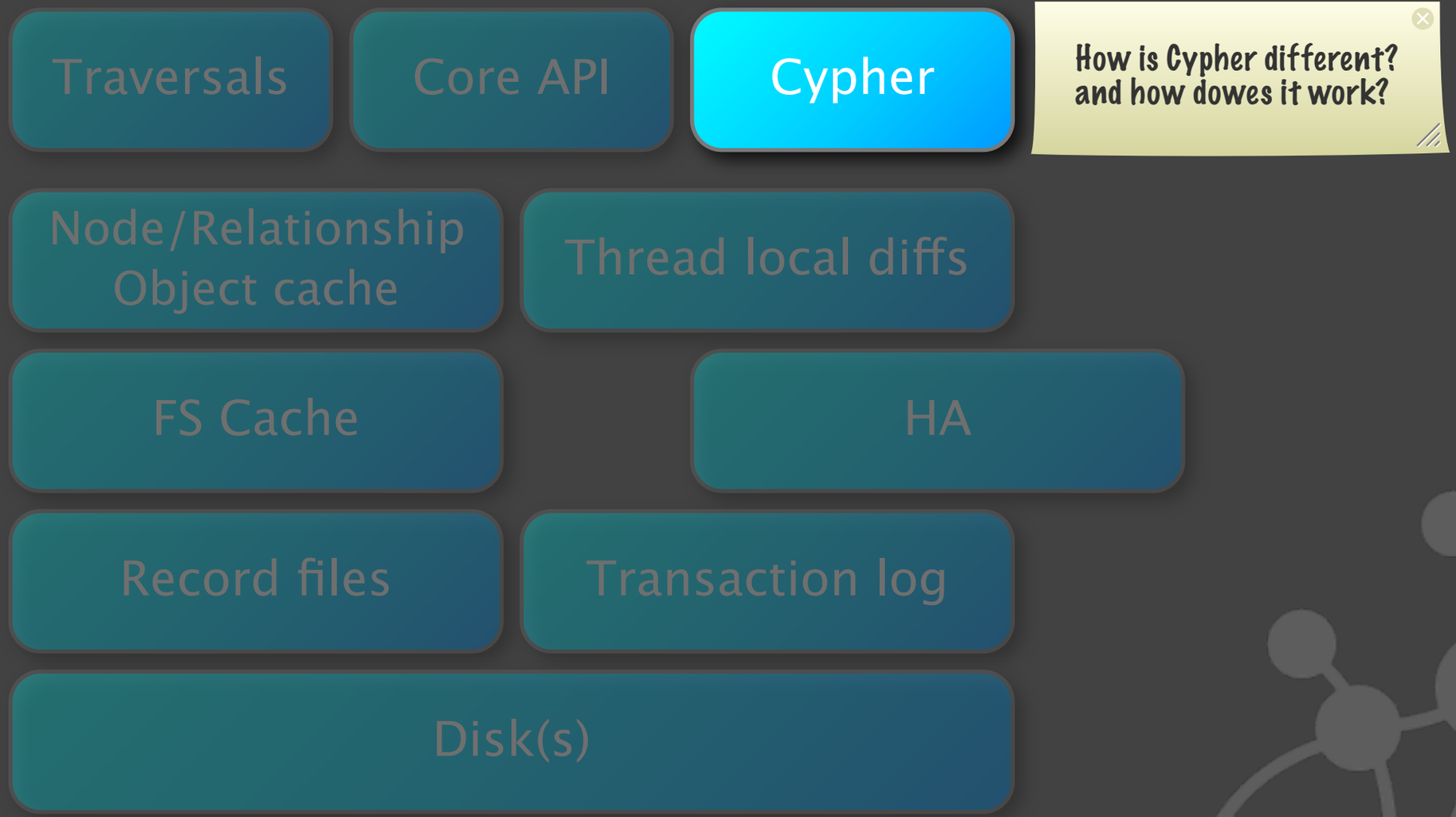
- ***Uniqueness*** *level*: policy for when it is ok to revisit a node that has already been visited
- Implemented on top of the Core API

More on Traversals

This is what happens under the hood.

- Fetch node data from cache - non-blocking access
  - `If` not in cache, retrieve from storage, into cache
    - `If` region is in FS cache: ***blocking*** but short duration access
    - `If` region is outside FS cache: ***blocking*** slower access
- Get relationships from cached node
  - `If` not fetched, retrieve from storage, by following chains
- **`Expand`** relationship(s) to end up on next node(s)
  - The relationship knows the node, no need to fetch it yet
- **`Evaluate`**
  - possibly emitting a **`Path`** into the result set
- Repeat

Outline


Traversals
Core API
Cypher
How is Cypher different?
and how dowes it work?
Node/Relationship
Object cache
Thread local diffs
FS Cache
HA
Record files
Transaction log
Disk(s)

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

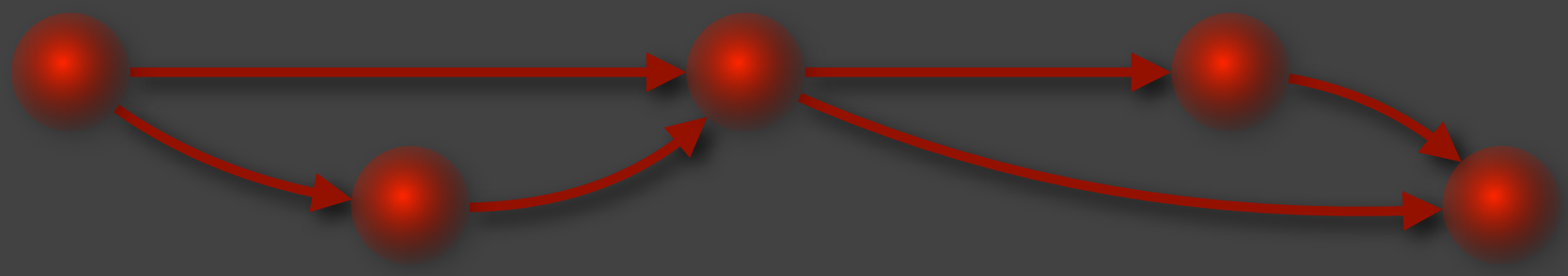
```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

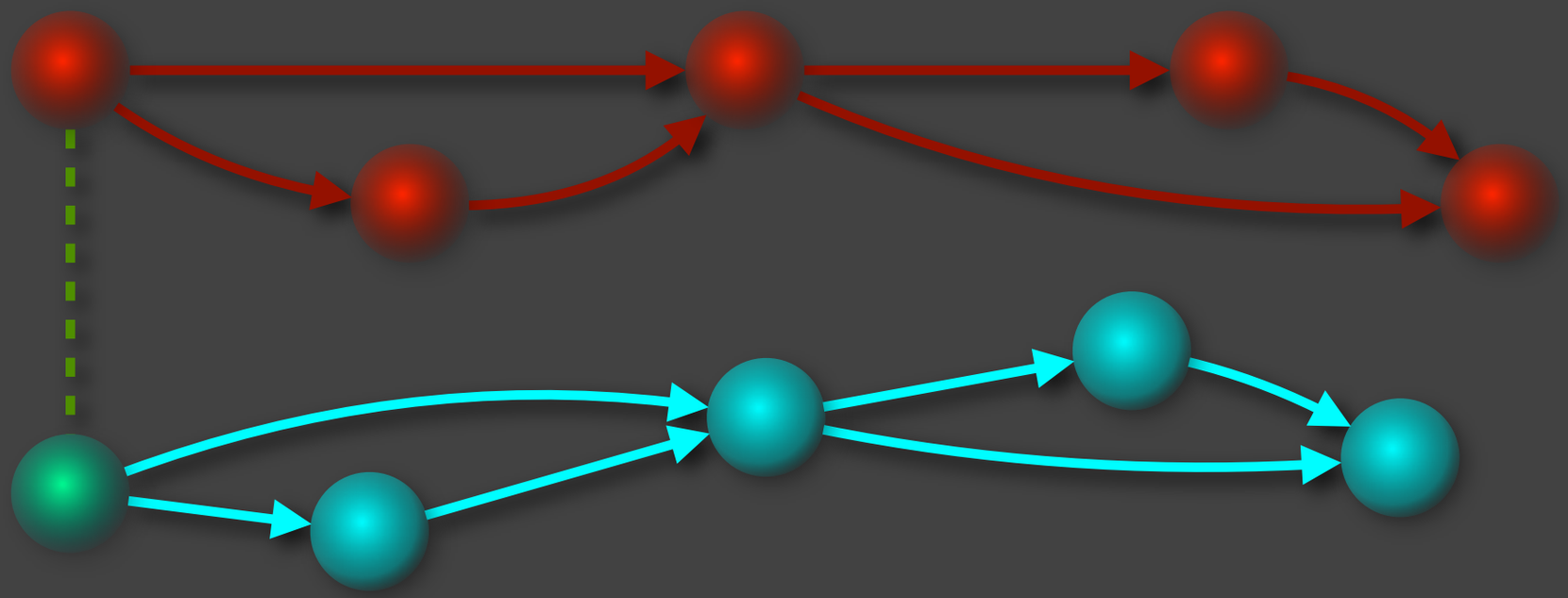


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

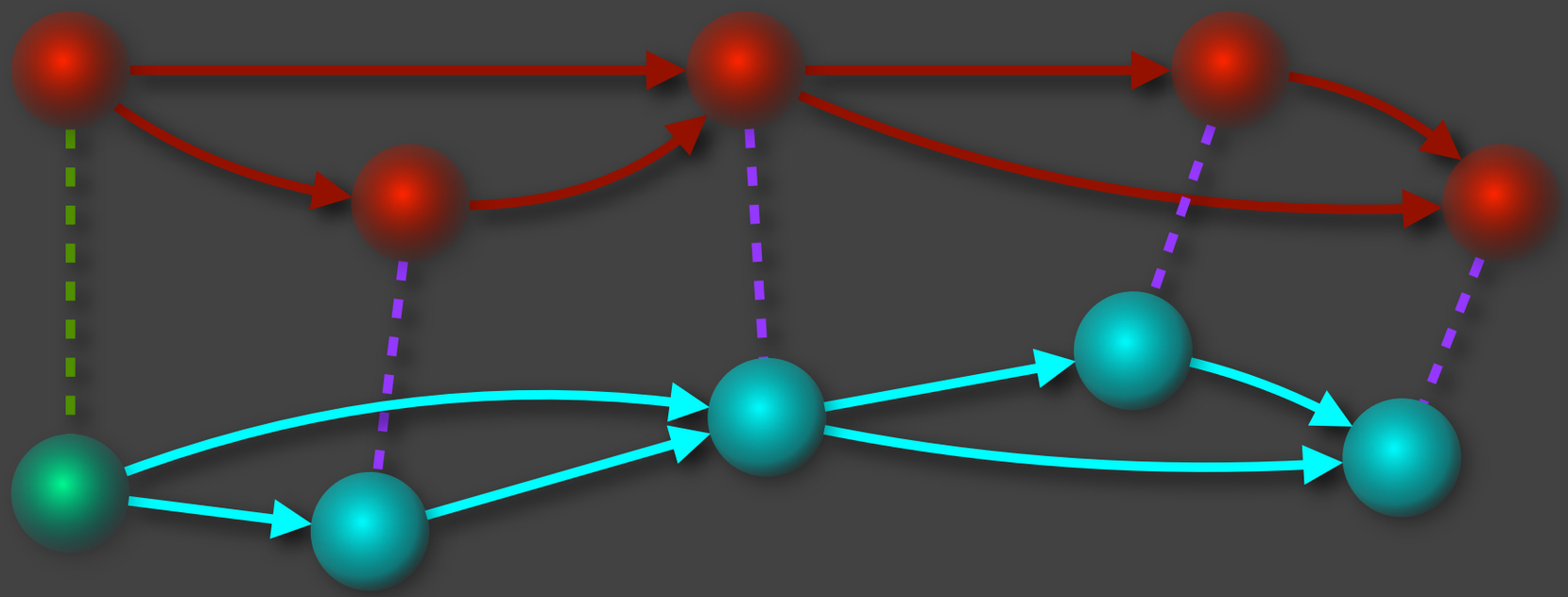


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

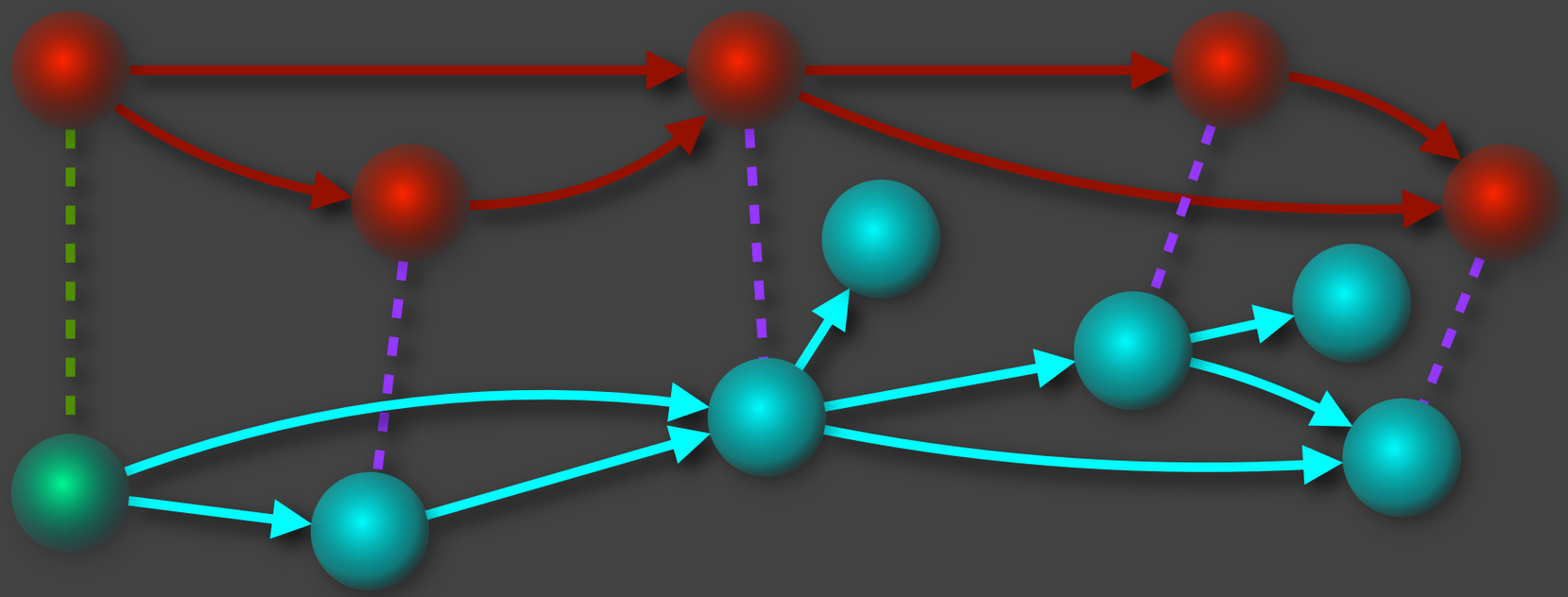


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

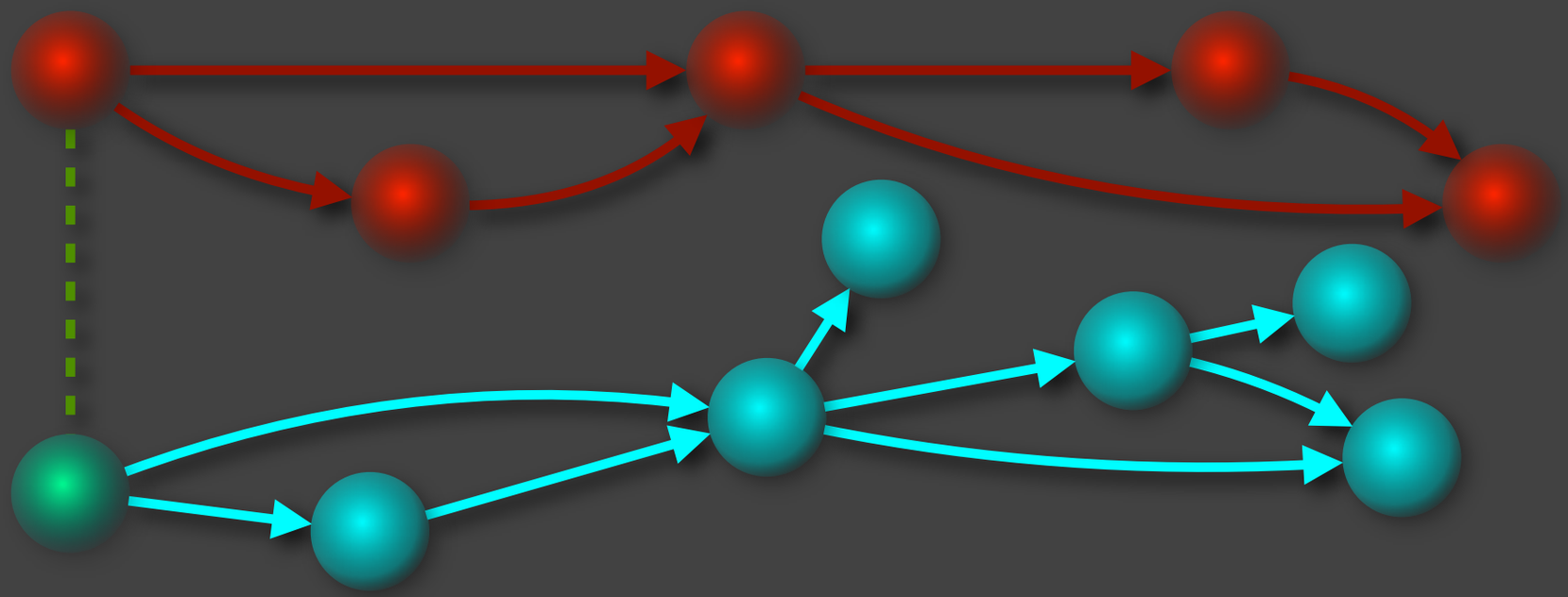


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

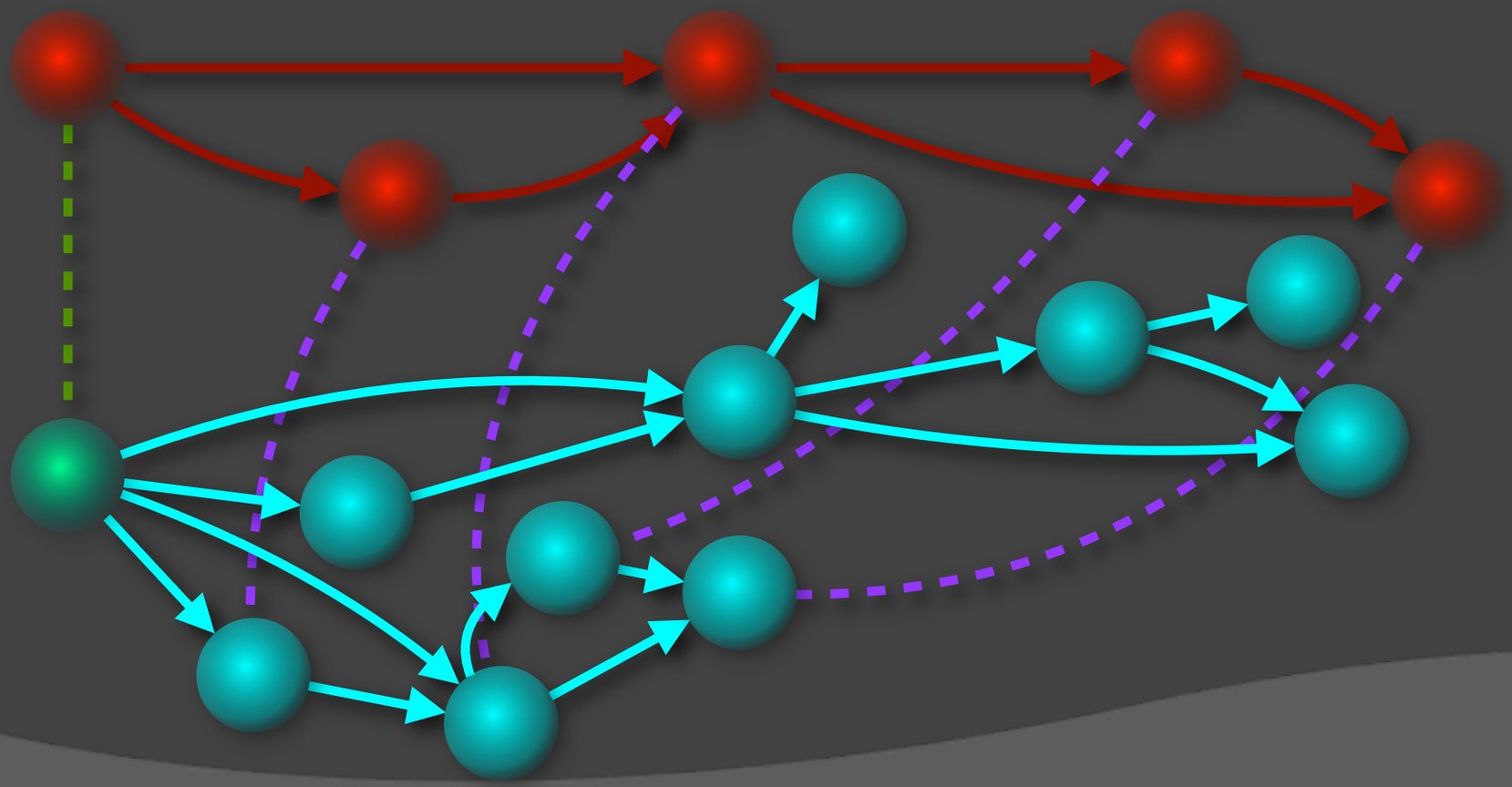


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

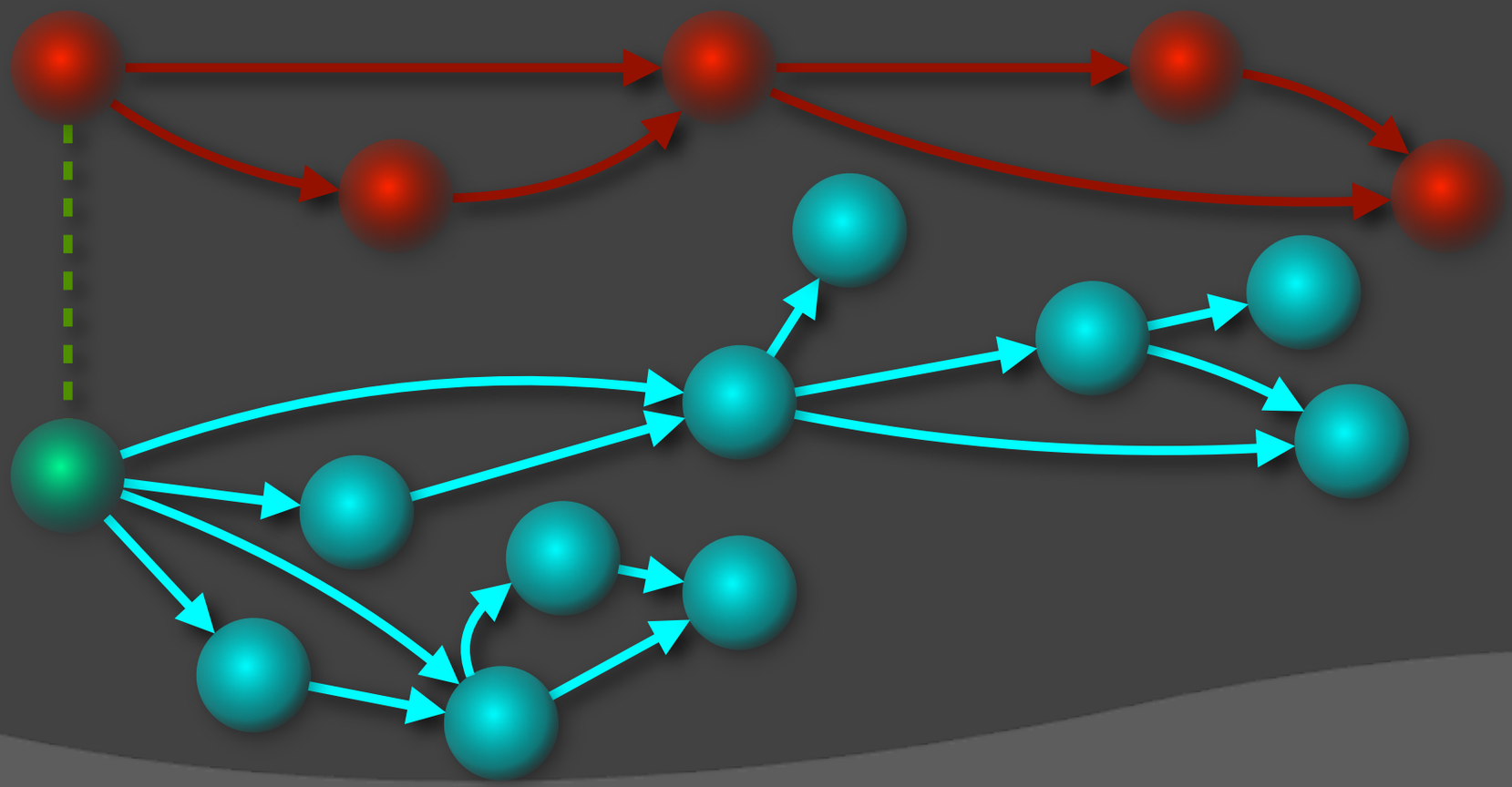


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```

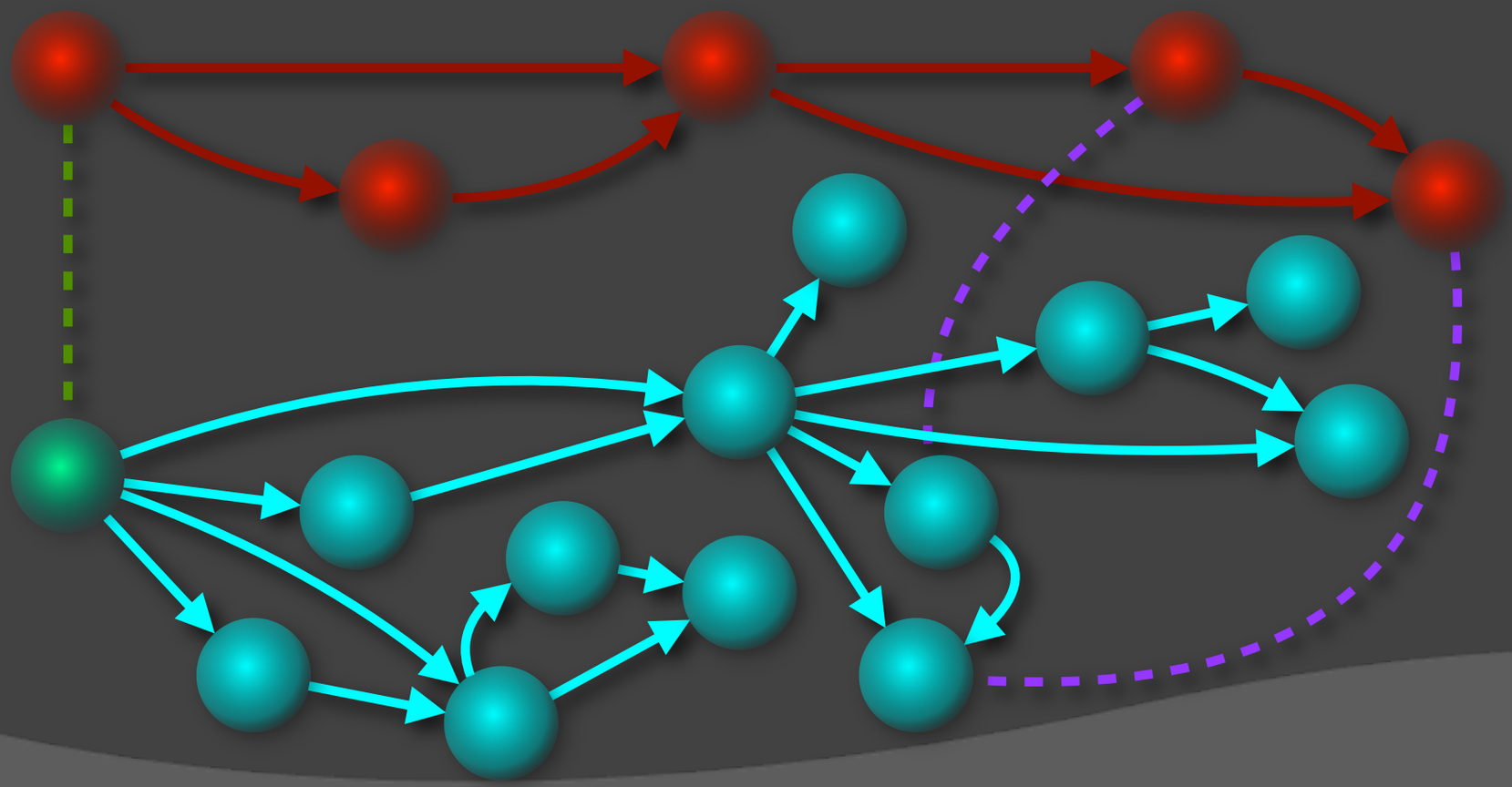


Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

Cypher - Just convenient traversal descriptions?

- Builds on the same infrastructure as Traversals - **`Expanders`**
  - but not on the full Traversal system
- Uses graph pattern matching for traversing the graph
  - Recursive matching with backtracking

```
START x=... MATCH x-->y, x-->z, y-->z, z-->a-->b, z-->b
```



Red: pattern graph
Blue: actual graph
Green: start node
Purple: matches

What about gremlin?

- gremlin is a third party language, built by Marko Rodriguez of Tinkerpop (a group of people who like to hack on graphs)
- Originally based on the idea of using xpath to describe traversals:

```
./HAS_CART/CONTAINS_ITEM/PURCHASED/PURCHASED
```

but bastardized to distinguish between nodes and relationships:

```
./outE[label=HAS_CART]/inV
 /outE[label=CONTAINS_ITEM]/inV
 /inE[label=PURCHASED]/outV
 /outE[label=PURCHASED]/inV
```

Traversals are close to xpath, which is why xpath like descriptions of traversals seemed like a good idea.

- xpath is not complete enough to express full algorithms, it needs a host language, gremlin originally defined its own.
  This changed Groovy as a more complete host language and abandoned xpath in favor of method chaining
  [replace '/' with '.']

Gremlin compared to Cypher

```
start me=node:people(name={myname})
  match me-[:HAS_CART]->cart-[:CONTAINS_ITEM]->item
  item<-[:PURCHASED]-user-[:PURCHASED]->recommendation
  return recommendation
```

- Cypher is declarative, describes ***what data to get*** - *its shape*
- Gremlin is imperative, prescribes ***how to get the data***
- Cypher has more opportunities for optimization by the engine
- Gremlin can implement pagerank, Cypher can't (yet?)

Outline

Traversals | Core API | Cypher

Node/Relationship Object cache | Thread local diffs

FS Cache | HA

Record files | Transaction log

Disk(s)

Transactions involve two parts:
The (thread local) changes being done by an active transaction,
and the transaction replay log for recovery.

Transaction Isolation

- Mutating operations are not written when performed
- They are stored in a thread confined transaction state object
- This prevents other threads from seeing uncommitted changes from the transactions of other threads
- When Transaction.finish() is invoked the transaction is either committed or rolled back
- Rollback is simple: discard the transaction state object

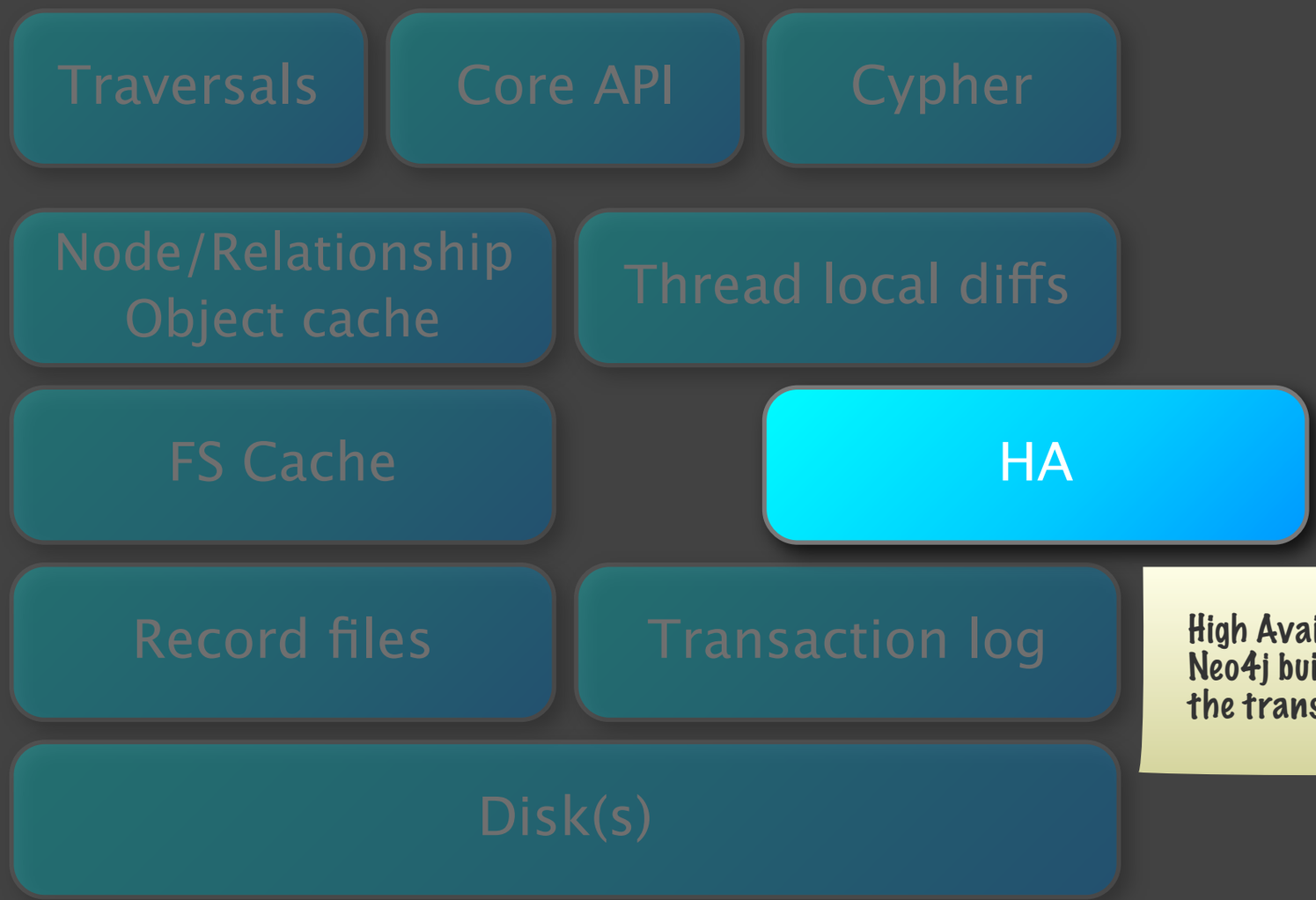
Transactions & Durability

- Commit is:
  - Changes made in the transaction are collected as commands
  - Commands are sorted to get predictable update order
    - This prevents concurrent readers from seeing inconsistent data when the changes are applied to the store
  - Write changes (in sorted order) to the transaction log
  - Mark the transaction as committed in the log
  - Apply the changes (in sorted order) to the store files

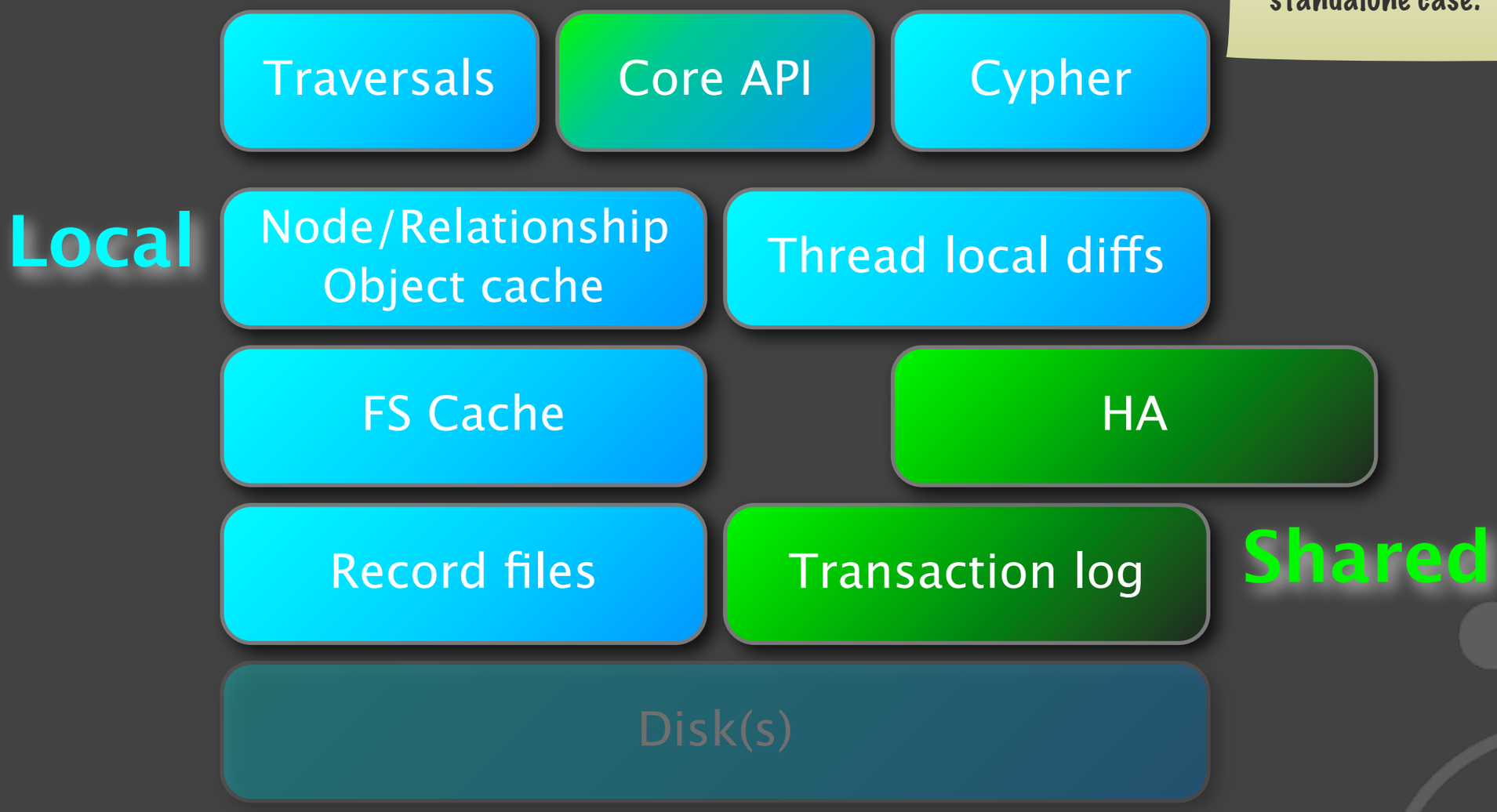
Recovery

- Transaction commands dictate state, they don't modify state
  - i.e. `SET property "count" to 5`
  - rather than `ADD 1 to property "count"`
- Thus: Applying the same command twice yields the same state
- Recovery simply replays all transactions since the last safe point
- If ***tx A*** mutates ***`node1.name`***, then ***tx B*** also mutates ***`node1.name`*** that doesn't matter, because the database is not recovered until all transactions have been replayed

Outline


Traversals
Core API
Cypher
Node/Relationship Object cache
Thread local diffs
FS Cache
HA
Record files
Transaction log
Disk(s)
High Availability in Neo4j builds on top of the transaction replay

Outline



The transaction logs are shared between all instances in an High Availability setup, all other parts operate on the local data just like in the standalone case.

- HA - the parts to it:
  - Based on streaming transactions between servers
  - All transactions are committed through the master
    - Then (eventually) applied to the slaves
      - Eventuality defined by the update interval
        or when synchronization is mandated by interaction
  - When writing to a slave:
    - Locks coordinated through the master
    - Transaction data buffered on the slave
      applied first on the master to get a txid
      then applied with the same txid on the slave

Creating new Nodes and Relationships

- New Nodes/Relationships don't need locks, so they don't need a transaction synced with master until the transaction commit
- They do need an ID that is unique and equal among all instances
- Each instance allocates IDs in blocks from the master, then assigns them to new Nodes/Relationships locally
- This batch allocation can be seen in (Enterprise) WebAdmin as Node/Relationship counts jumping in steps of 1000

HA synchronization points

- Transactions are uniquely identified by monotonically increasing ID
- All Requests from slave send the current latest txid on that slave
- Responses from master send back a stream of transactions that have occurred since then, along with the actual result
- Transaction is replayed just like when committed / recovered
- Nodes/Relationships touched during this application are evicted from cache to avoid cache staleness
- Transaction commands are only sorted when created, before stored/transmitted, thus consistency is preserved during all application phases

Locking semantics of HA

- To be granted a lock the slave must have the latest version of the Node/Relationship it is trying to lock
  - This ensures consistency
  - The implementation of "Latest version of the Node/Relationship" is "Latest version of the entire graph"
  - The slave must thus sync transactions from the master

Master election

- Each instance communicates/coordinates:
  - its latest transaction id (including the master id for that tx)
  - the id for that instance
  - (logical) clock value for when the txid was written
- Election chooses:
  1. The instance with highest txid
  2. ***IF*** *multiple*: The instance that was master for that tx
  3. ***IF*** *unavailable*: The instance with the lowest clock value
  4. ***IF*** *multiple*: The instance with the lowest id
- Election happens when the current master cannot be reached
  - Any instance can choose to re-elect
  - Each instance runs the election protocol individually
  - Notify others when election chooses new master
- When elected, the new master broadcasts to all instances, forcing them to bind to the new master

Thank you for listening!

Tobias Lindaaker

Hacker @ Neo Technology

tobias@neotechnology.com
twitter: @thobe, #neo4j (@neo4j)
web: neo4j.org neotechnology.com
my web: thobe.org