DC GraphDB Meetup - May 2013

- 630 Refreshments & Networking
- 700 Announcements
- 705 What's new in Neo4j 2.0 Cypher
Neo4j Internals
- Q & A, hanging out

geek cred?

Overview 2.0

- Labels!
- Indexing overhaul (to support labels)
- Cypher syntax (to support labels and indexes)
- Transactional endpoint for Cypher
- Other cool stuff... CASE/WHEN, MERGE, UNION

Labels

- Labels are basically types for nodes
- You can have more than one per node
- A lot of the Graph/Object mappers store properties for “node type”--in 2.0 they can use labels
- Compelling enough alone to try 2.0, IMO

Labels, cont.

- Use to specify the new automatic indexes (no more config file editing/restarting)
- Can be set from Cypher, in a query
- Great for separating out subgraphs
- Must be strings--stored similar to string properties

Indexing overhaul

- New "schema" indexes to support labels
- Initially, again, in Lucene
- Similar to legacy auto-indexes in behavior, but can be created on the fly (in Cypher!!)
- Queries can take advantage of them automatically -- WHAT?! (we'll look at that)

Index performance improvements?

- Out of the box faster for most things than 1.9 "legacy" indexes, due to lack of two-phase commit
- Experimentation with MapDB
- Only 2 simple java classes required to implement your own index; see Michael Hunger's blog post for MapDB: `http://bit.ly/19cYv3m`

Cypher syntax

- Old:

  ```
  START a= node:node_auto_index(“name:*”)
  MATCH a-[:ACTED_IN]->m
  RETURN a.name, m.title;
  ```

- New:

  ```
  MATCH a:Actor-[:ACTED_IN]->m:Movie
  RETURN a.name, m.title;
  ```

INDEX syntax

- Old:

```
START a=node:node_auto_index(“name:Andrés”)
RETURN a;
```

- New:

```
MATCH a:Actor
WHERE a.name = ‘Andrés’
RETURN a
```

- Automatically uses the index, based on the WHERE clause!
- You can still use “legacy” indexes if you want

INDEX syntax

- Old:

```
START a=node:node_auto_index(name=“Keanu Reaves”),
          d=node:node_auto_index(name=“Lana Wachowski”)
MATCH a-[:ACTED_IN]->m<-[:DIRECTED]-d
RETURN a,m,d;
```

- New:

```
MATCH a:Actor-[:ACTED_IN]->m:Movie<-[:DIRECTED]-d:Director
WHERE a.name = ‘Keanu Reaves’ and d.name=‘Lana Wachowski’
USING INDEX d:Director(name)
RETURN a,m,d;
```

- It will pick an index if it doesn’t know which will be better. But you can hint if you know where you want to start.

CREATE INDEX syntax

- Old:
  Oh wait, there's no syntax to create indexes...
- New:
  CREATE INDEX on :Actor(name);
- Creates in background, returns from the command immediately.

Transactional REST API

Hard to illustrate in a single slide...
but it’s awesome.

- Begin (POST): http://localhost:7474/db/data/transaction
- Execute (POST): http://localhost:7474/db/data/transaction/6
- Commit (POST): http://localhost:7474/db/data/transaction/6/commit
- Rollback (DELETE): http://localhost:7474/db/data/transaction/6
- Begin/Commit (POST) commands in a transaction, similar to batch: http://localhost:7474/db/data/transaction/commit

New: CASE/WHEN

- Case/When:

  ```
  MATCH n
  RETURN CASE
              WHEN n.age < 20 THEN ‘1-20‘
              WHEN n.age < 30 THEN ‘twenties‘
              END as age_range
  ```

- Also works with: CASE n.age WHEN 20 THEN... form
- Great for grouping results or massaging data, just like in SQL.

UNION

- Similar to SQL UNION (removes dups):

  ```
  MATCH a:Actor
  RETURN a.name as name
  UNION
  MATCH d:Director
  RETURN d.name as name
  ```

- Also UNION ALL supported (doesn’t remove dups)

MERGE

- UPSERT for subgraphs or single nodes
- Declarative ON CREATE, ON UPDATE--like SQL triggers
- Can use labels to create unique nodes!
  MERGE (:Actor {name:”Keanu”});

Let’s get internal

- Files on disk
- MMIO
- Object Caching
- Wishlist for 2.1

Disk files

- Fixed size records mixed with dynamic block chained records--often inlined data if possible
- Fixed: Node, Relationship, RelTypeIndex, PropertyIndex
- Dynamic: Property, Dynamic, Strings, Labels, etc.

MMIO settings

- For fixed size records, easy to calculate MMIO needs
- Ideally you can fit the whole nodes/rels files in MMIO, based on planned # nodes/rels.
- See default settings in neo4j.properties (at the top of the file)
- Extremely important using batch-import, or your import might take an order of magnitude longer than expected

On disk: nodes - history

1.4 - <=2011 - 9 bytes, 2^32 max* nodes/rels/props

InUse	firstRelId	firstPropId
1 byte	4 bytes	4 bytes

*really 2^32 - 2, which is roughly 4.3B

1.5 - 1.9 - 9 bytes, 2^35 max* nodes/rels, 2^36 props

InUse	firstRelId	firstPropId
1 byte	4 bytes	4 bytes

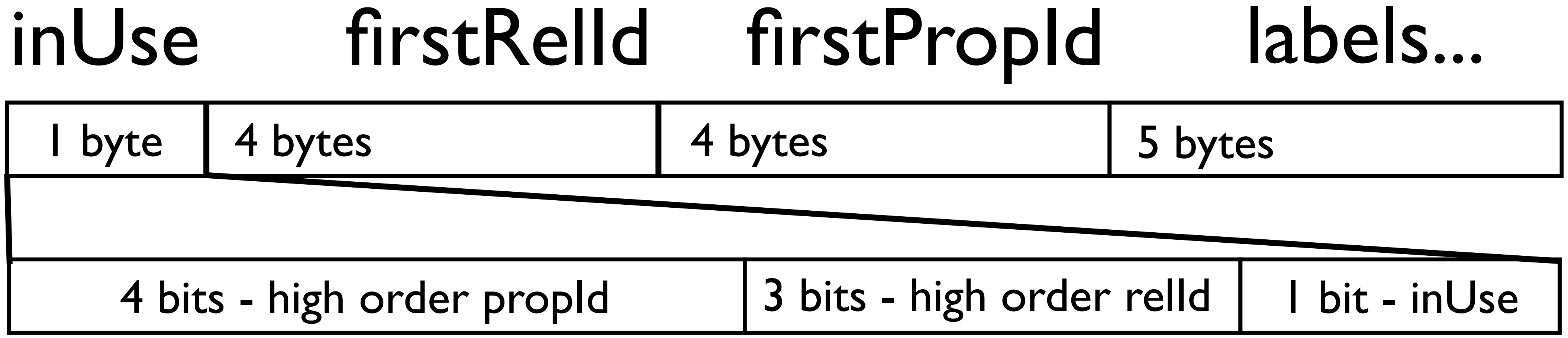
4 bits - high order propId	3 bits - high order relId	1 bit - InUse

*really 2^35 - 2, which is roughly 34.5B max nodes/rels,
and 2^36 - 2, which is roughly 68.7B max properties

DRAWING NOT TO SCALE

On disk: nodes

2.0 - 2013 - 14 bytes:
2^35 max* nodes/rels, 2^36 props, dynamic labels



inUse firstRelId firstPropId labels...

1 byte	4 bytes	4 bytes	5 bytes

4 bits - high order propId	3 bits - high order relId	1 bit - inUse

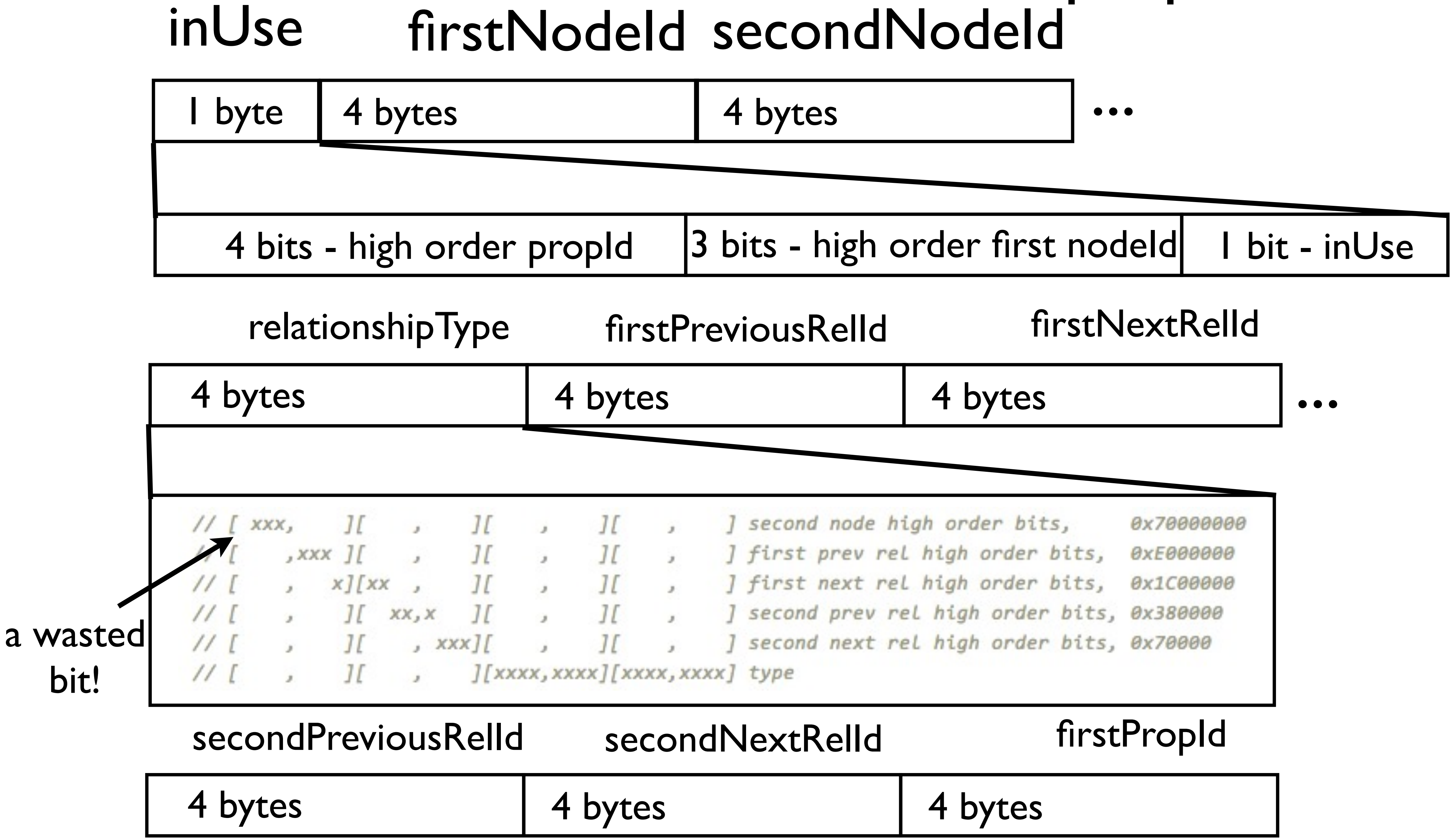
*really 2^35 - 2, which is roughly 34.5B max nodes/rels,
and 2^36 - 2, which is roughly 68.7B max properties

this is the same as 1.9

On disk: relationships

1.5 - 2.0 - 33 bytes

2^35 max* nodes/rels, 2^36 props



```
// [ xxx,    ][    ,    ][    ,    ][    ,    ] second node high order bits,     0x70000000
// [    ,xxx ][    ,    ][    ,    ][    ,    ] first prev rel high order bits,  0xE000000
// [    ,   x][xx  ,    ][    ,    ][    ,    ] first next rel high order bits,  0x1C00000
// [    ,    ][  xx,x   ][    ,    ][    ,    ] second prev rel high order bits, 0x380000
// [    ,    ][    , xxx][    ,    ][    ,    ] second next rel high order bits, 0x70000
// [    ,    ][    ,    ][xxxx,xxxx][xxxx,xxxx] type
```

DRAWING NOT TO SCALE

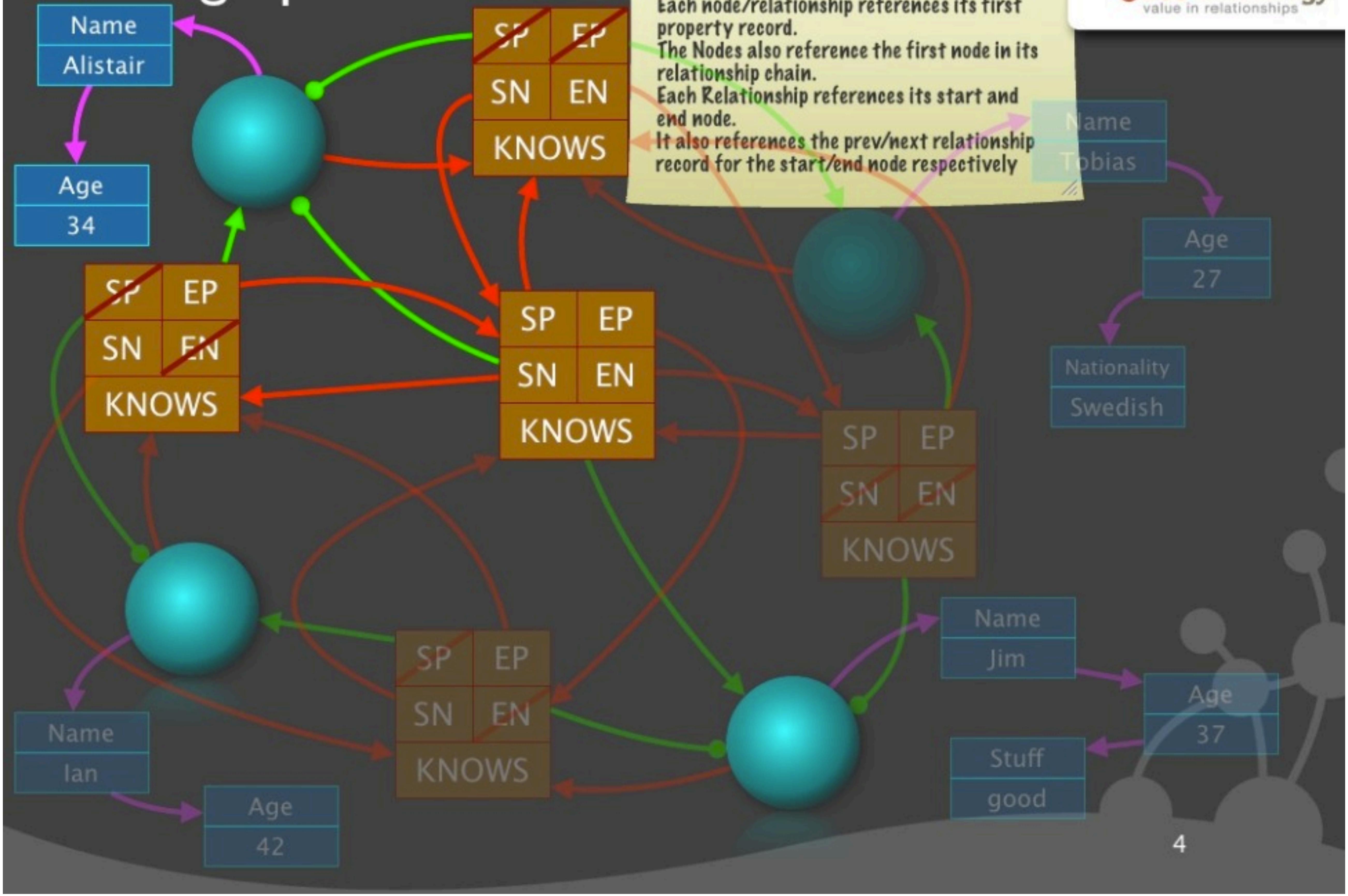
To increase limits...

- Store files would need to grow per node/rel/prop by a number of bytes, at least with the current disk layout
- Most use cases don’t need more than the current limits
- Could probably be hacked out in a few days if necessary (creating a new layout for the store file)

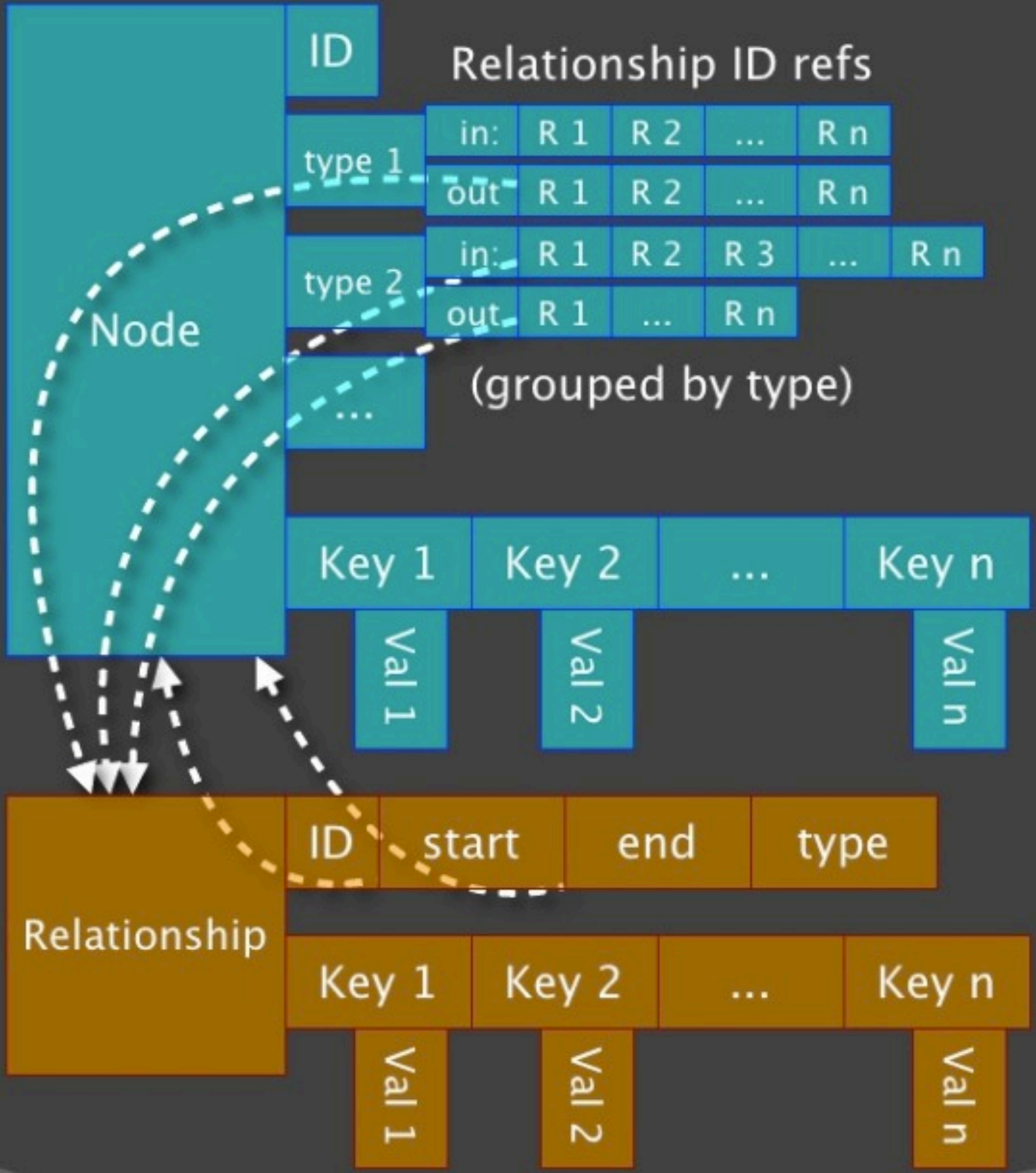
Your graph on disk

neotechnology
value in relationships

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



What we put in cache



Node

ID

Relationship ID refs

type 1

in: R 1 R 2 ... R n

out R 1 R 2 ... R n

type 2

in: R 1 R 2 R 3 ... R n

out R 1 ... R n

...

(grouped by type)

Key 1 Key 2 ... Key n

Val 1 Val 2 Val n

Relationship

ID start end type

Key 1 Key 2 ... Key n

Val 1 Val 2 Val n

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.

Cool things that might be in 2.1 - performance

- Relationship store optimized for large numbers of relationships per node (not linked list)
- Make disk store more like the object cache, and remove the object cache--direct MMIO (save RAM, avoid using heap)
- Cypher cost-based query optimization
- ID packing and delta compression to save on disk space and in turn RAM usage