

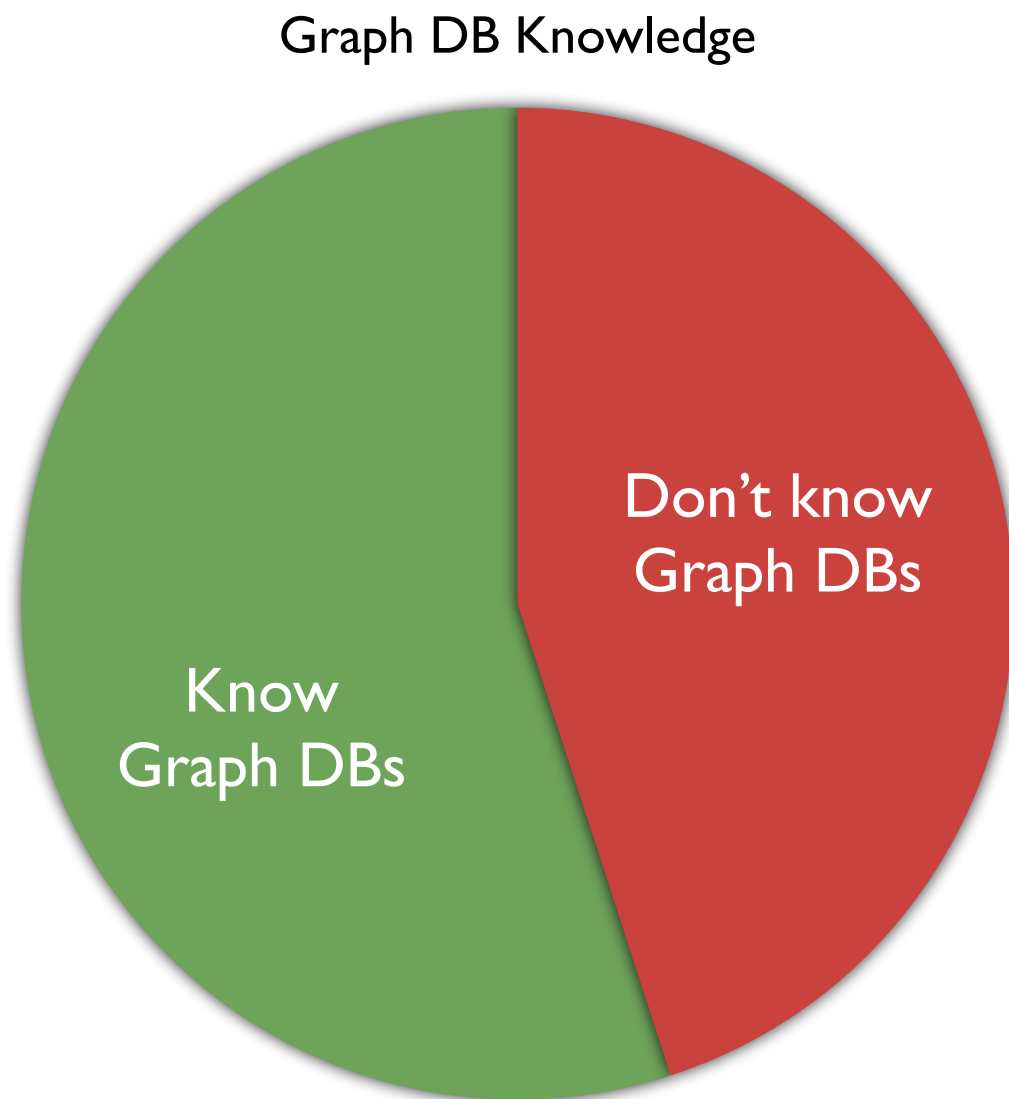
Inside Neo4j's Graph Query Engine

Stefan Plantikow

- Neo4j
- Graphs: Benefits, Model, Querying
- Cypher Query Language
- Query Engine (+ some scala snippets)
- Joint work from the Neo4j Cypher Team

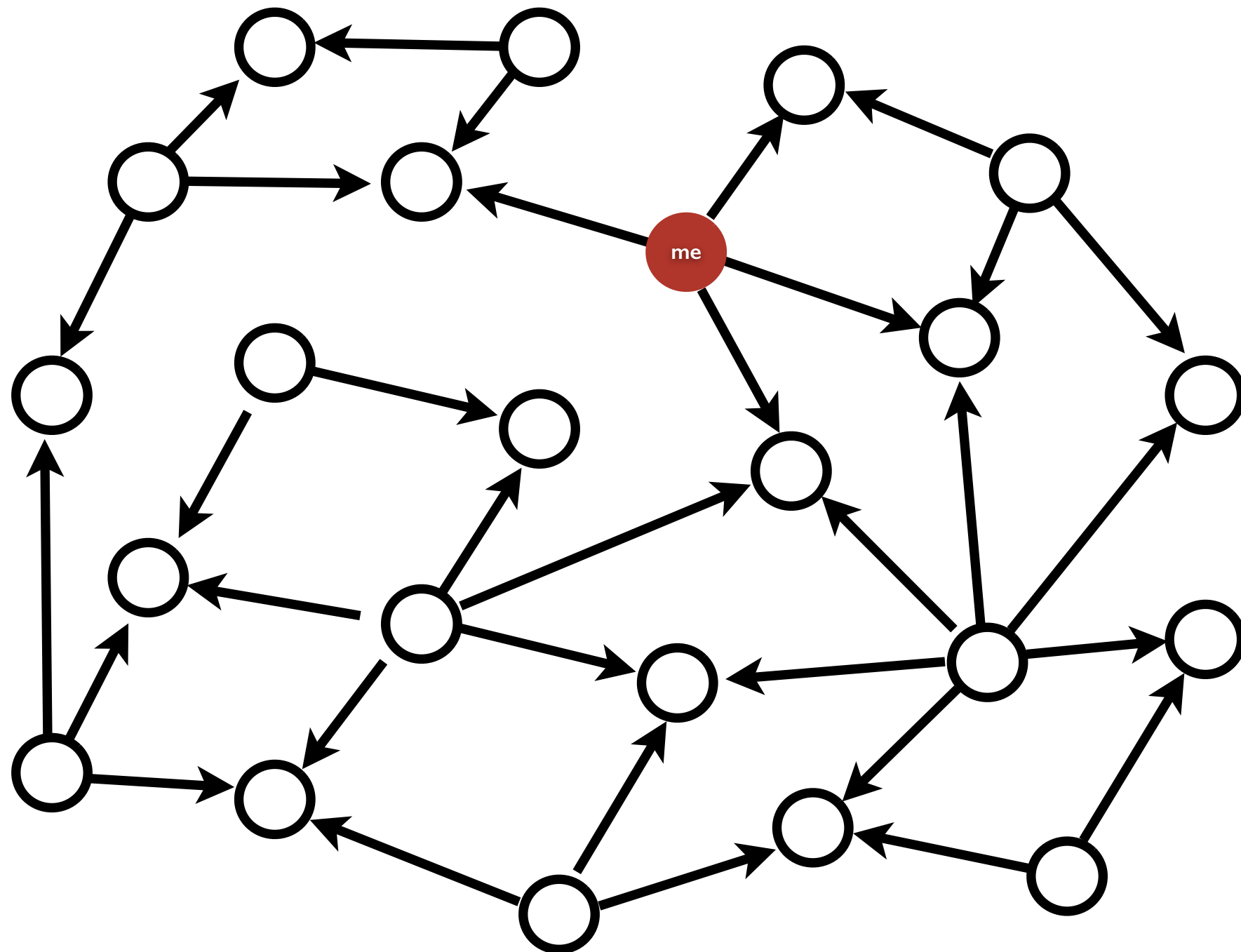
Neo4j is a Graph Database

Graphs != Charts

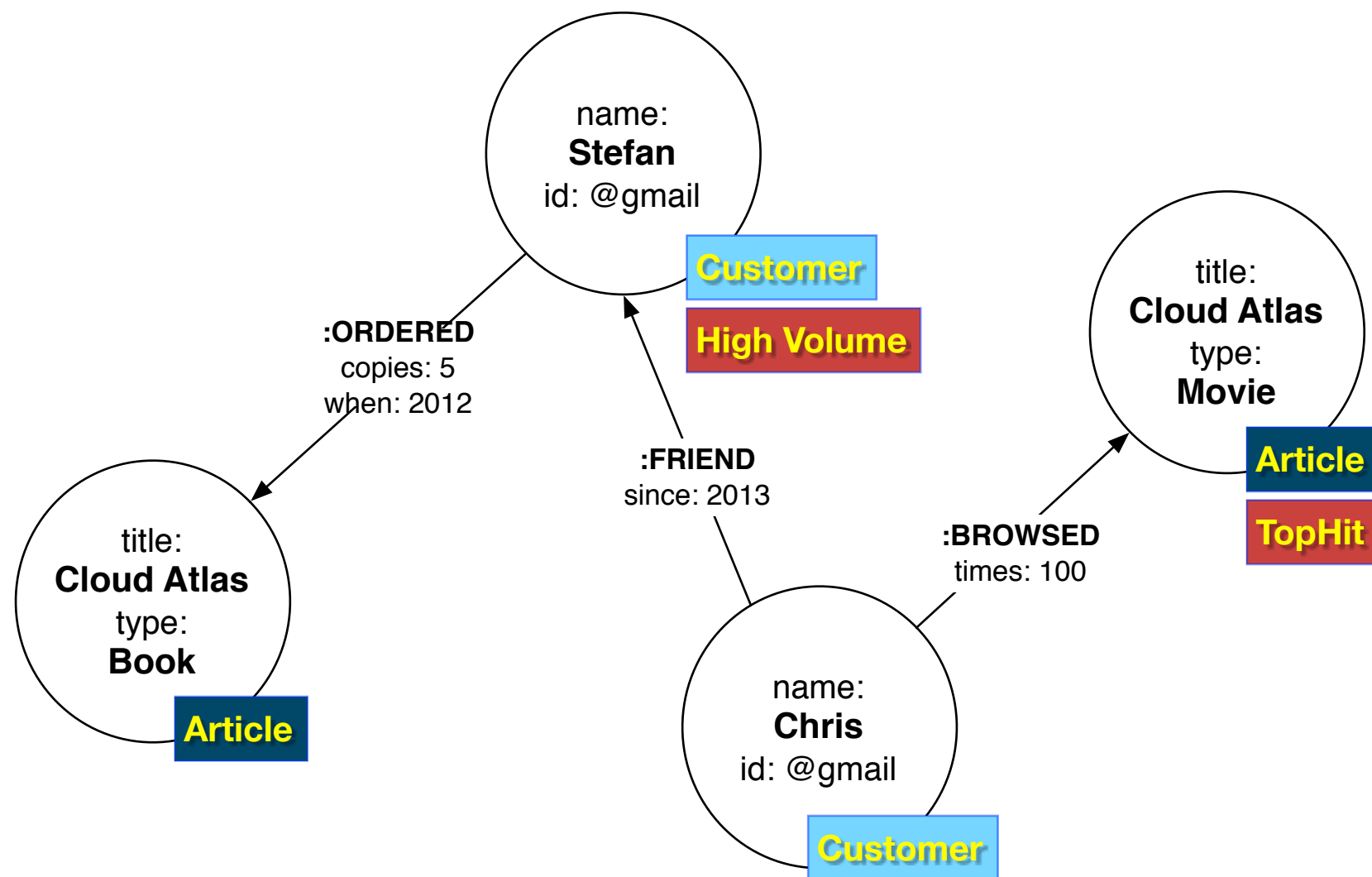


Charts != Graphs

Directed Graph



Labeled Property Graphs



Graphs

- Nodes („things“)
- Relationships
 - Math: multigraphs, hypergraphs, probabilistic graphs, ...
- Graph DBs: Property Graphs (Directed Multigraph with Properties)
 - Key-Value Properties: name, last login, number of posts, ...
 - Label: Person, Article, Sale, ...
 - Indices: by name, by ID, ...

Neo4j

- Dual-Licensed: Open Source (A(GPL)) / Commercial
- Transactional (ACID / Read-Committed)
- Server with Web-UI
- High Availability
- Tooling: Shell, Backup, Monitoring (JMX), ..
- Interfaces: REST API, Core API, Cypher, Language Bindings

Neo4j Implementation

- Kernel: Java
Web-UI: Javascript
 - Few dependencies
 - kernel.jar: 3 MB
 - DirectByteBuffer
 - GC-resistant custom caches
 - ...
- Cypher (Query Language):
Scala 2.10
 - Runtime w. kernel integration
 - Enterprise Modules
 - HA
 - Online Backup
 - ...

Neo4j Browser

The screenshot displays the Neo4j Browser interface. The top navigation bar shows the URL `localhost:7474/browser/`. The left sidebar contains the 'Neo4j 2.0-SNAPSHOT' header, a star icon, and sections for 'Labels' (Movie, Animal, coder, Fairy, expert, Person, User, Bird), 'Relationships' (LIKES, ACTED_IN, DIRECTED, PRODUCED, WROTE, FOLLOWS, REVIEWED), 'Properties' (released, title, prop, name, test, summary, tagline, born, id, roles, age, sex, rating), and 'Database' (Location: `/Users/stepn/Current/neo4j/community/server/neo4j-home/data`, Size: 1.32 MB).

The main area shows a Cypher query: `match (n {title: 'The Matrix'})--(m) return n, m;`. The results are displayed as a graph with 27 nodes and 24 relationships. The nodes are colored by type: orange for Movie, purple for Director, teal for Actor, pink for Reviewer, yellow for Producer, and blue for Person. The graph shows a central orange node (343) connected to several blue nodes (345, 344, 341, 300, 349, 348, 347) via 'ACTED_IN' relationships. There are also two orange nodes (343, 174) at the bottom.

Below the graph, two more Cypher queries are shown:

```
CYPHER match (n {title: 'The Matrix'})-->(r) return n, r;
```

Returned 0 rows in 137 ms

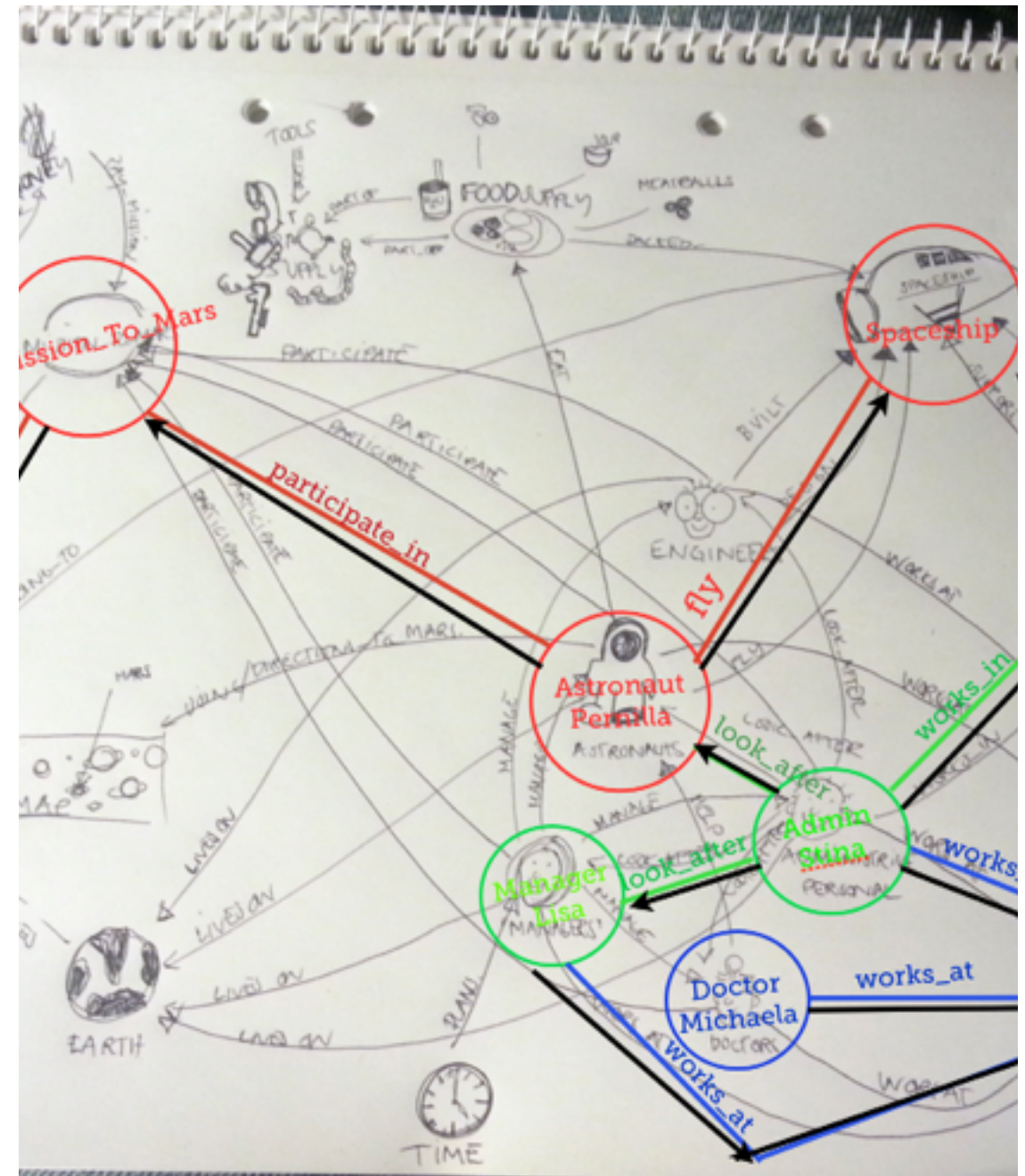
```
CYPHER match (n {title: 'The Matrix'}) return n;
```

After Installation: <http://localhost:7474>

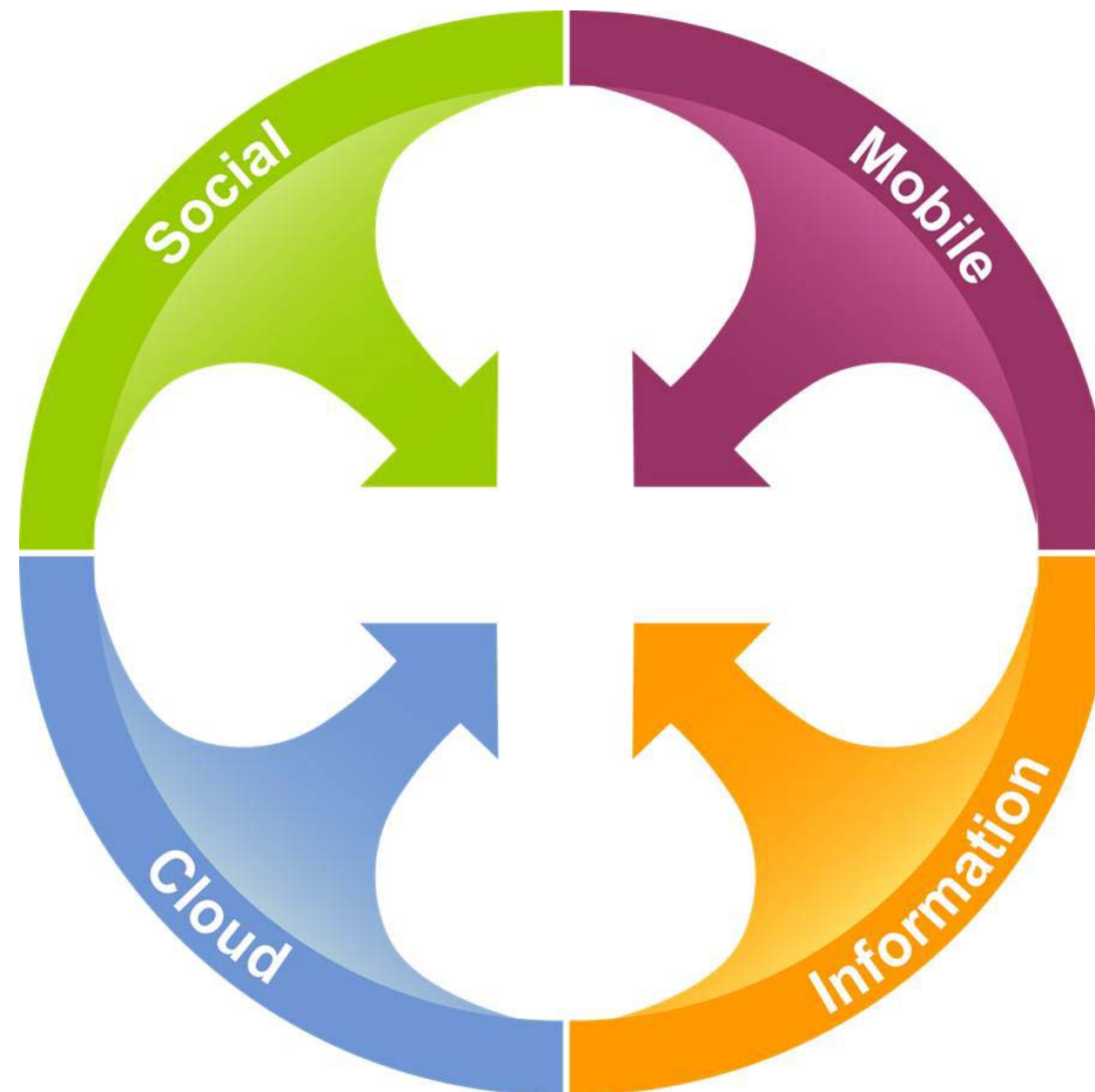
Why Graphs

Graph Benefits

- Natural Model
- Whiteboard Friendly
- Match OOP
- Extensible and Uniform
- Ease Data Integration



Graphs are Everywhere



Gartner



„25% of enterprises will be using graph
databases by 2017“

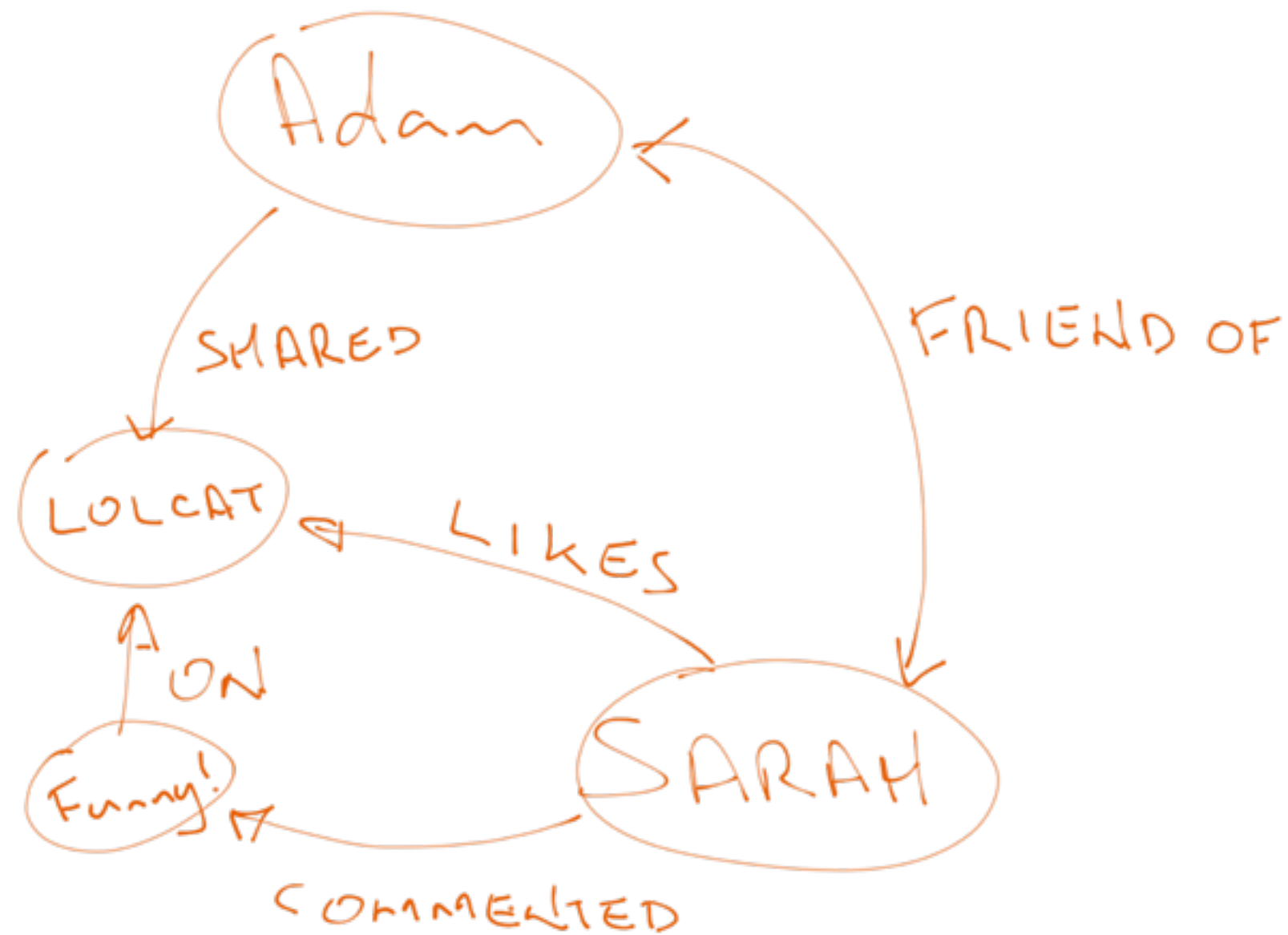
– Forrester Research

Use-Cases

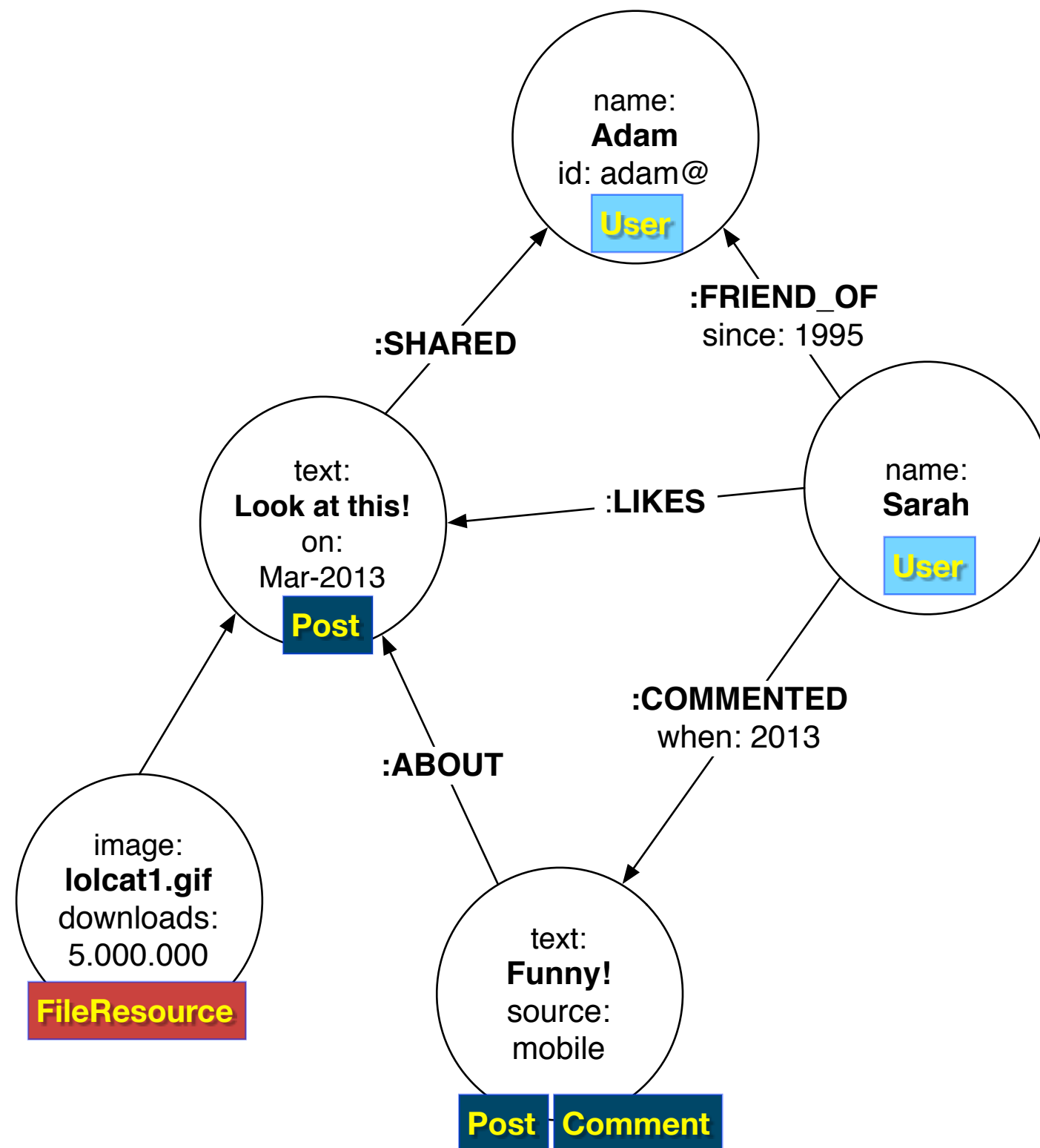
- Social Networks & Recommendations
- Geospatial
- Infrastructure as a Service
- Business Intelligence
- Content Management
- Access Control
- Bioinformatics
- Genealogy
- Telco
- Finance
- ...

Modeling with Graphs

From whiteboard...



...to data model



Cypher Graph Query Language

Querying Graphs

- for a user with the name „Stefan“ find all articles that have been browsed by any of his friends at least 4x

return them sorted by times browsed

- find instances of a given pattern

and compute a result

Querying Graphs

- MATCH (n {name: „Stefan“),
MATCH (n)-[:FRIEND]->(f),
(f)-[r:VIEWED]->(a)

RETURN a, collect(r) as score
ORDER BY score

- find instances of a given pattern
and compute a result

Querying Graphs

- Find Patterns

- Describe matching nodes and relationships and how they should be connected
- Node und Relationship identified via entity ID or primary key
- Indices
Exact, full text, geospatial

- Build result

- Sort
- Aggregate
- Combine & Filter & Transform

Cypher

- Neo4j's

// Cypher

```
MATCH (actor:Actor)-[:ACTS_IN]->(movie:Movie)
RETURN actor.name, movie.title
```

- Declarative

// SQL

```
SELECT Person.name, Movie.title
FROM Person
```

```
JOIN Actor on
```

```
Person.person_id = Actor.person_id
```

```
JOIN Movie on
```

```
Movie.movie_id = Actor.movie_id
```

- Language

- or:
SQL for Graphs

Cypher: Basic Example

- Declarative query language with SQL-like clause syntax
- Visual graph patterns
- Tabular results

// get node

```
MATCH (a:Person {id: 0}) RETURN a
```

// return friends

```
MATCH (a:Person {id: 0})-->(b) RETURN b
```

// return friends of friends

```
MATCH (a:Person {id: 0})--()--(c) RETURN c
```


Cypher: Filter and Sort

- Filter using predicates in WHERE
- Aggregate, sort, limit

// lookup all nodes as 'n', constrained to name 'Stefan'

```
MATCH (n:People) WHERE n.name='Stefan' RETURN n
```

// filter nodes where age is less than 30

```
MATCH (n:People) WHERE n.age < 30 RETURN n
```

// filter and aggregation using a regular expression

```
MATCH (n:People) WHERE n.name =~ "Mat.*" RETURN count(n)
```

// find nodes with a property and return first 3 found

```
MATCH (n:People) WHERE has(n.name) RETURN n LIMIT 3
```

// find the 5 oldest people

```
MATCH (n:People) RETURN n ORDER BY n.age LIMIT 5
```

Cypher: Graph Queries

- Variable Length Path
- Shortest Path

// books liked by friends of friends

```
MATCH (stefan:People {name: „Stefan“}),  
      (stefan)-[:FRIEND..2]->(friend)-[:LIKE]->(b:Book)  
RETURN DISTINCT b
```

// shortest path between two people

```
MATCH  
  p = shortestPath((martin:People)-[*..15]-(oliver:Person))  
WHERE  
  martin.name = 'Martin Sheen' AND oliver.name = 'Oliver Stone'  
RETURN p
```

Cypher: Updating Nodes and Relationships

// create node

```
CREATE (a:People {name: 'Andres'})
```

// create relationship between bound nodes

```
CREATE (stefan)-[:KNOWS]->(andres)
```

// ensure unique node exists and set title

```
MERGE (b:Book {id: 123}) SET b.title = 'Cloud Atlas'
```

// match and update

```
MATCH (n:People) WHERE n.age = 34 SET n.age = 35
```

// find node and delete it

```
MATCH (n:People) WHERE n.name = 'Dr. Evil' DELETE n
```

Cypher: Much more

- Handling path sets
- Functional expressions: Extract, Filter, Reduce
- Optional Match (Outer Join)
- Profiling
- ...

Query Planning

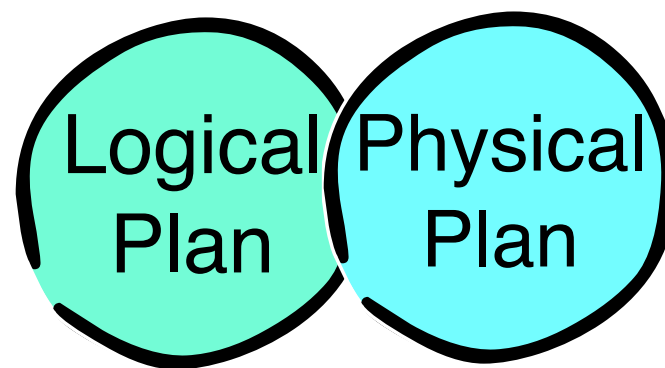
Query Planning

- From Cypher to Results
- Need to build operator tree
- And run it

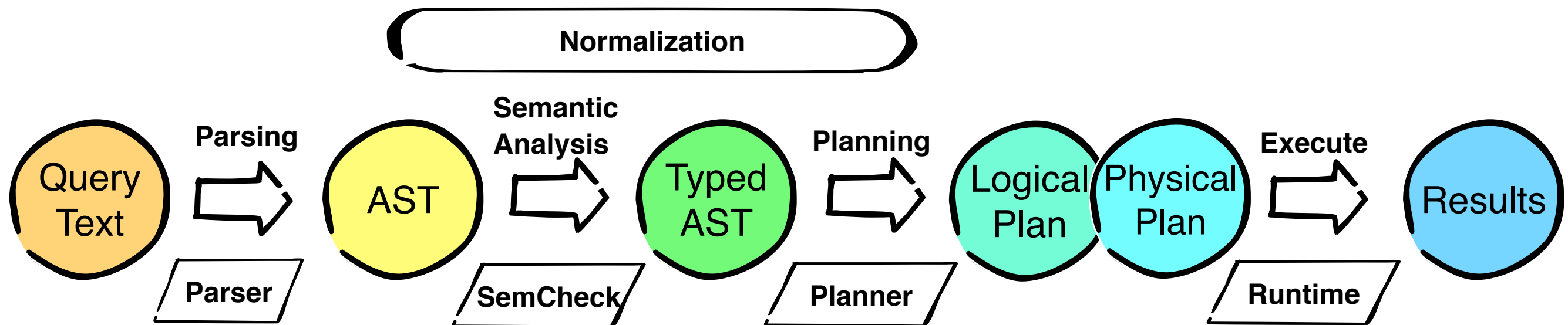
```

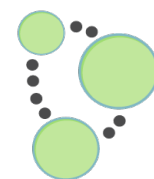
MATCH (a:Person)-[:FRIEND*..2]->(b)
WHERE a.name = „Stefan“ AND
      a.city <> b.city AND
RETURN a.name AS a,
      b.name AS b, b.city AS city

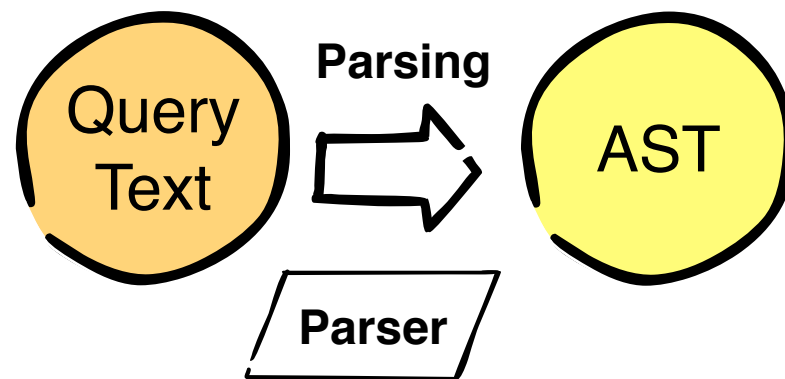
```



a	b	city
„Stefan“	„Luli“	Beijing
„Stefan“	„Francesca“	Rome
„Stefan“	„Jack“	New York City

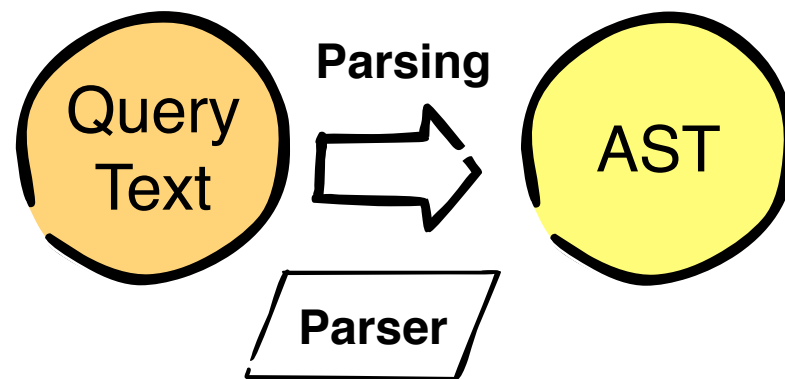


 **Neo4j** Enterprise Graph Database Server



- We use parboiled (not yet version 2)
- Packrat Parsing Library in scala
- Output is AST (+ back references to input string)
- Grammar has 140+ ASTNode classes
- Example: `DISTINCT split("abba", "b") => ["a", "", "a"]`

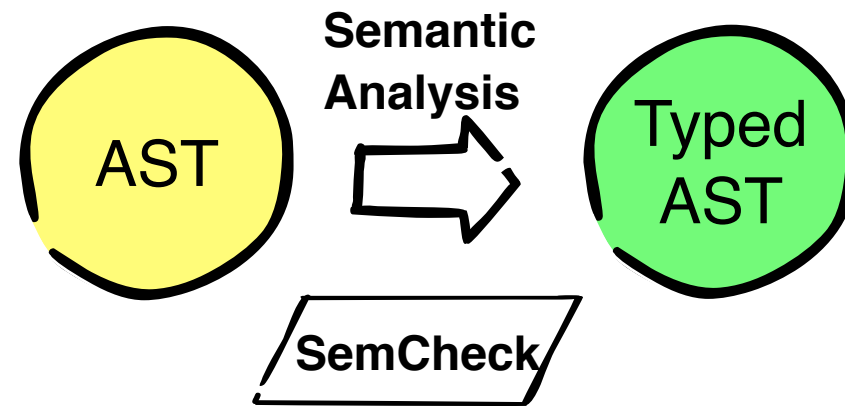
```
FunctionInvocation(  
  FunctionName("split")_,  
  distinct = true,  
  args = Vector(StringLiteral("abba") _, StringLiteral("b") _)  
)_  
  
pprintToString(expr) should equal("DISTINCT split(1, 2)")  
}
```



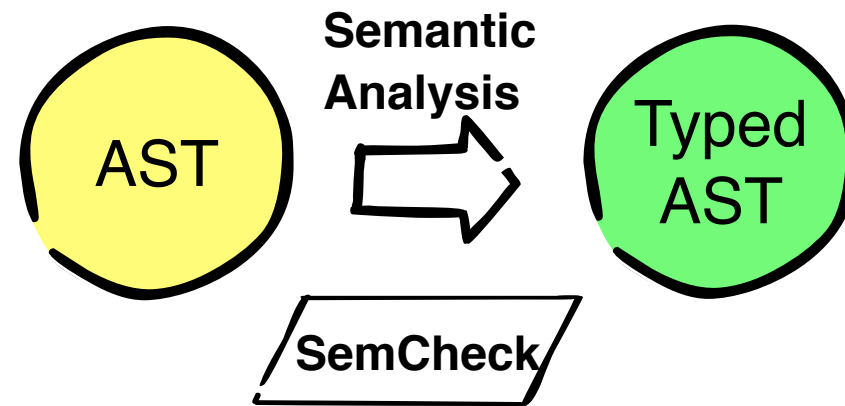
- Parsing rules manipulate value stack
- Rule firing has stack effect
(similar to how concatenative languages work)

```
def Create: Rule1[ast.Clause] = rule("CREATE") (  
  group(keyword("CREATE UNIQUE") ~~ Pattern) ~~>> (ast.CreateUnique(_))  
  | group(keyword("CREATE") ~~ Pattern) ~~>> (ast.Create(_))  
)
```

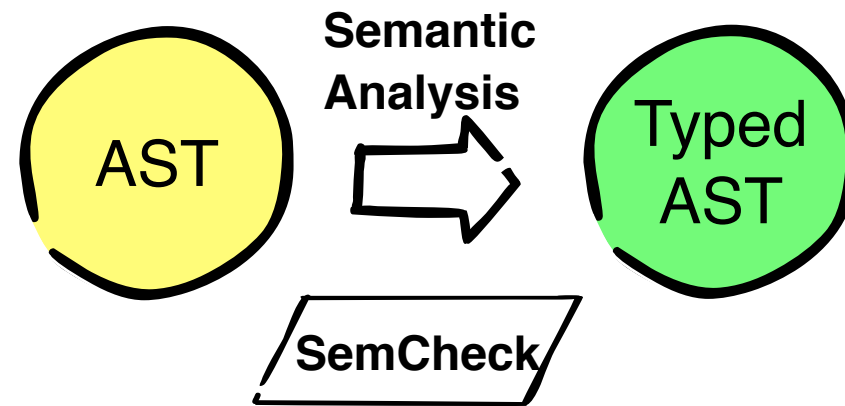
```
private def FunctionInvocation: Rule1[ast.FunctionInvocation] = rule("a function") {  
  ((group(FunctionName ~~ "(" ~~  
    (keyword("DISTINCT") ~ push(true) | EMPTY ~ push(false)) ~~  
    zeroOrMore(Expression, separator = CommaSep) ~~ ")"  
  ) ~~> (_.toIndexedSeq)) memoMatches) ~~>> (ast.FunctionInvocation(_, _, _))  
}
```



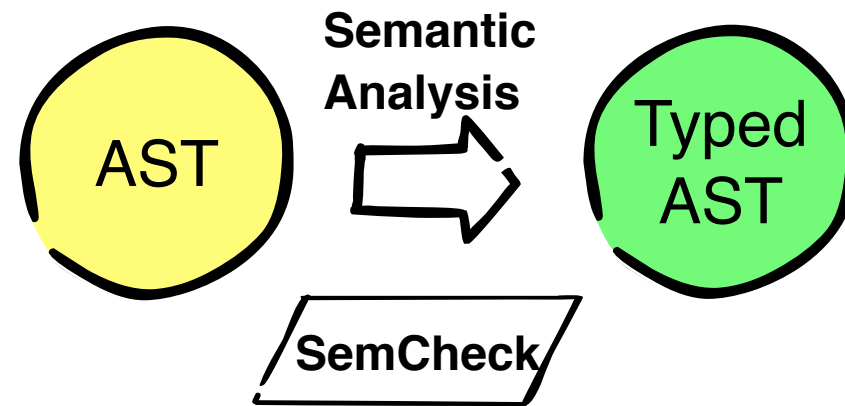
- SemCheck: Ensure the query „makes sense“, e.g.
 - Does not return many columns with same name
 - Relationships in CREATE are directed
 - Nicer error messages for difficult to parse conditions



- SemCheck: Type Checking
 - Primitives
 - Strings
 - Numbers (Integral and Floating)
 - Booleans
 - Collections<T>
 - Map<String, T>
 - Graph entities (Nodes, Relationships) are treated as Map<String, Any>
- Not always known: MATCH n RETURN n.prop
- Deferred to runtime (form of gradual typing)



- SemCheck: Implementation
 - Hand-rolled State Monad
 - Walk AST tree keeping track of scope and type information
 - Collect errors along the way



- SemCheck: Implementation
 - Walk AST tree keeping track of scope and type information
 - Collect errors along the way
 - Hand-rolled State Monad

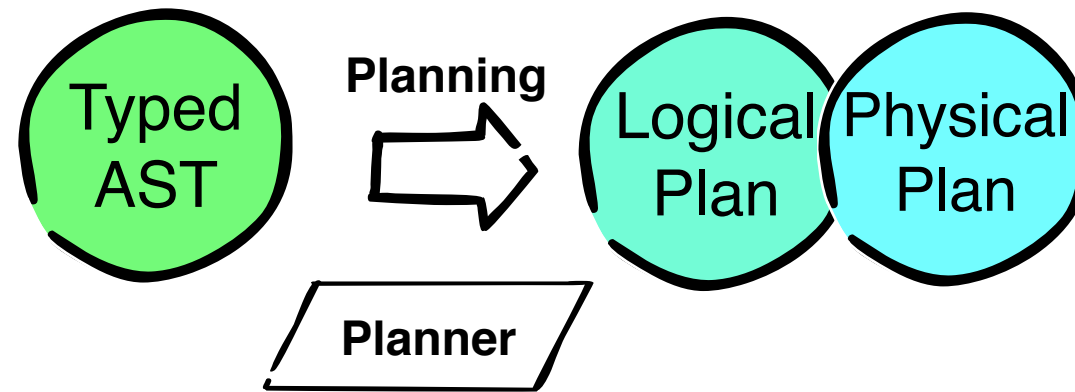
Normalization

- Rewriting AST Nodes into normal form
 - Expand aliases: $\text{RETURN } * \Rightarrow \text{RETURN } x \text{ AS } x, y \text{ AS } y$
 - Constant folding: $1+2*4 \Rightarrow 9$
 - Name anonymous pattern nodes: $\text{MATCH } () \Rightarrow \text{MATCH } (n)$
 - Inlining
 - ...
- Own rewriter framework
- Allows pattern matching and replacing of tree nodes (bottom-up, top-down)

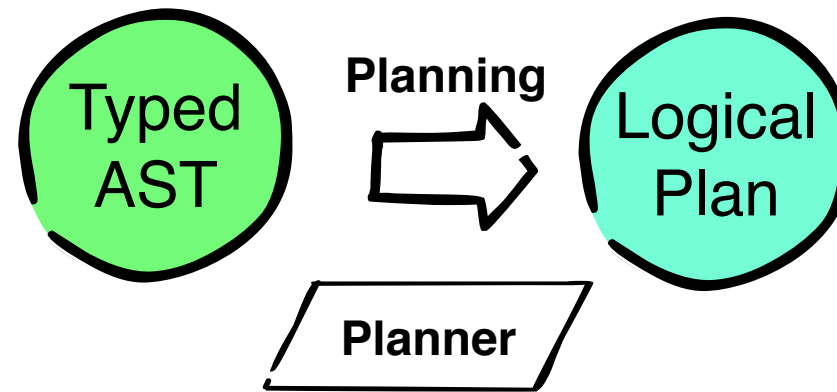
Normalization

MATCH (n) WHERE id(n) = 12 =>
MATCH n WHERE id(n) IN [12]

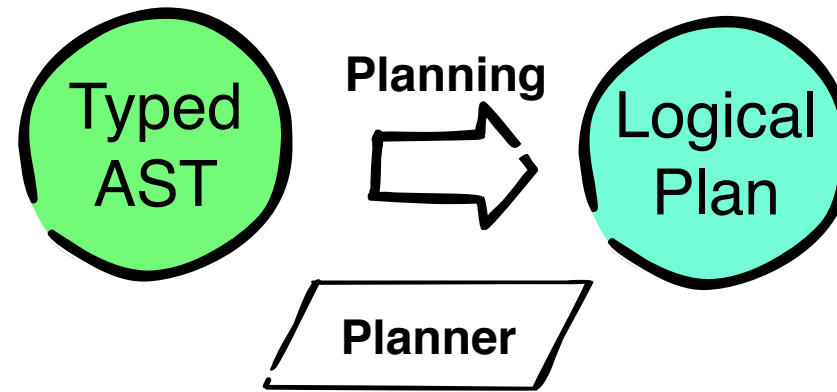
```
case object rewriteEqualityToInCollection extends Rewriter {  
  override def apply(that: AnyRef) = bottomUp(instance).apply(that)  
  
  private val instance: Rewriter = Rewriter.lift {  
    // id(a) = value  
    case predicate@Equals(func@FunctionInvocation(_, _, IndexedSeq(idExpr)), p@ConstantExpression(idValueExpr))  
      if func.function == Some(functions.Id) =>  
  
      In(func, Collection(Seq(idValueExpr))(p.position))(predicate.position)  
  
    // a.prop = value  
    case predicate@Equals(prop@Property(id: Identifier, propKeyName), p@ConstantExpression(idValueExpr)) =>  
      In(prop, Collection(Seq(idValueExpr))(p.position))(predicate.position)  
  }  
}
```

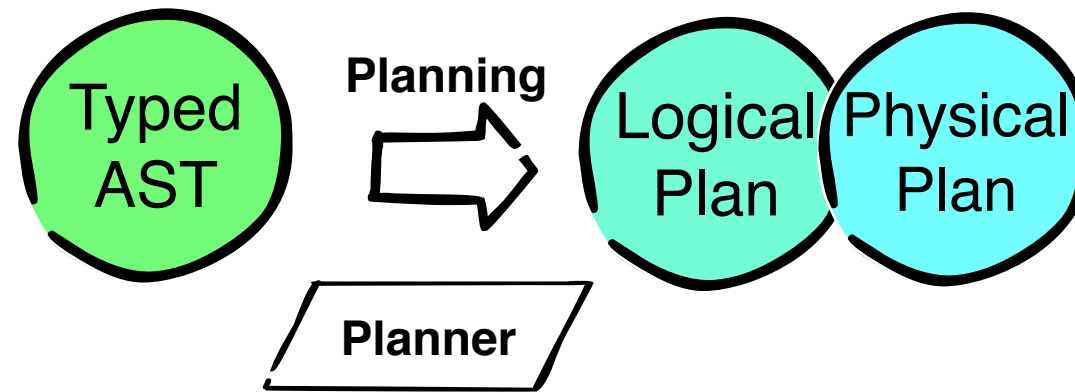
- Build semantic model from AST
- Turn semantic model into logical plan
- Turn logical plan into physical plan



- Build semantic model from AST
 - Which nodes?
 - Which relationships?
 - Which predicates?
 - How to return result
 - ...



- Construct a logical plan
 - Iteratively search space of candidate plans
 - Per iteration:
Gradually build new candidate plans
 - Use statistical cost model to distinguish between good and bad candidates („join ordering“)
 - Keep going until query solved by a good plan



- Logical Plan
 - Tree of operators
 - Similar to relational databases
 - Different operators
- Physical Plan
 - Choose physical implementation for logical operators

MATCH

```
(a:Artist), (b: Artist),  
(a)-[r:WORKED_WITH*..5 {year: 2014}]->(b)
```

RETURN

```
a.name AS name, collect(b) AS colleagues  
ORDER BY size(colleagues)
```

- Node a
- Node b
- Relationships r
- a is an Artist, b is an Artist, r.year = {year}, ...
- Projection with aggregation and sorting

MATCH

(a:Artist), (b: Artist),

(a)-[r:WORKED_WITH*..5 {year: 2014}]->(b)

RETURN

*

Compiler CYPHER 2.2-cost

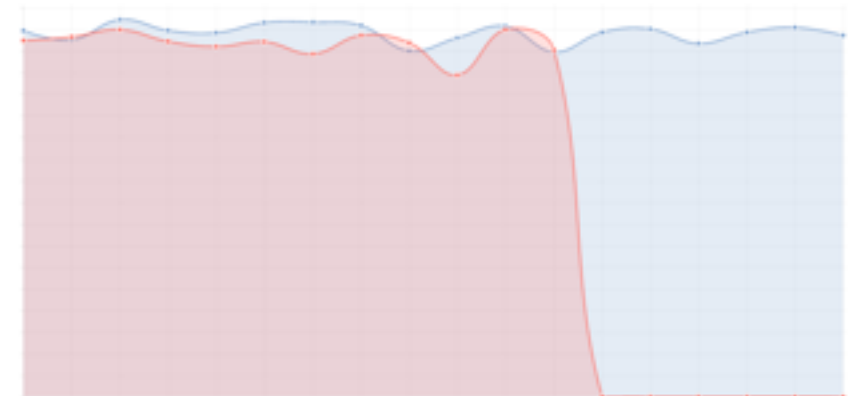
Filter

|

+Var length expand

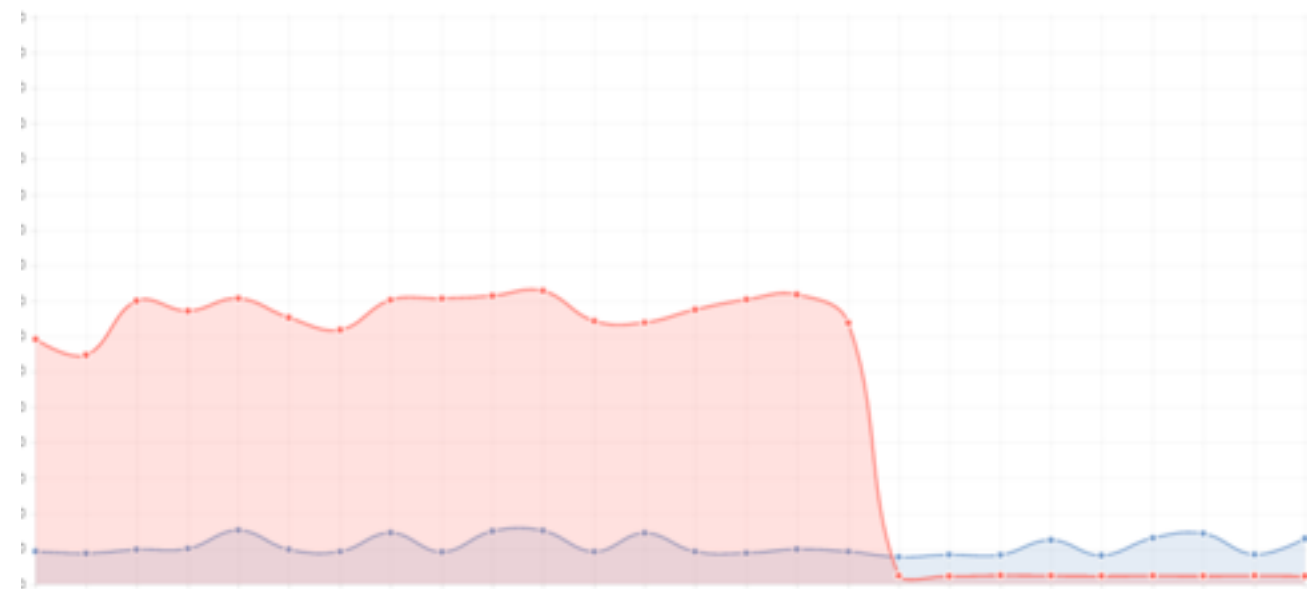
|

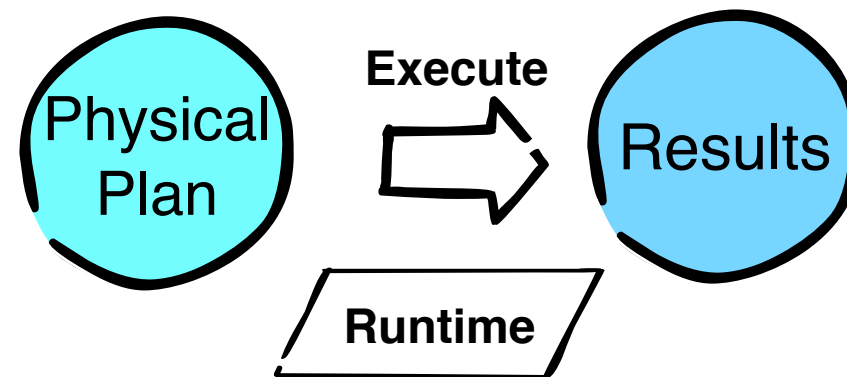
+NodeByLabelScan



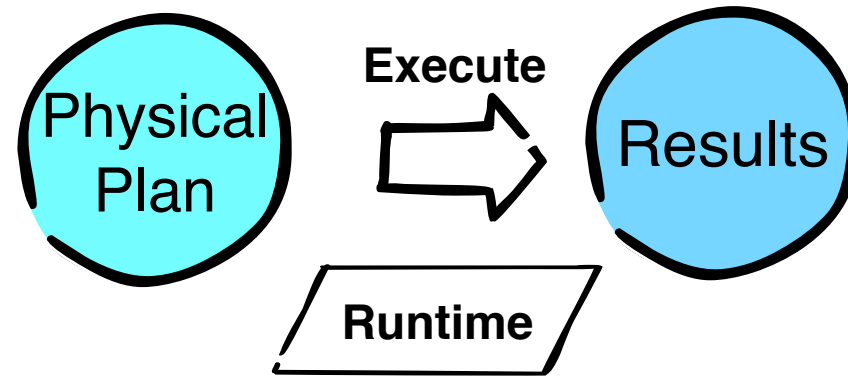
```
MATCH (a1:Album)
WHERE (:Artist)-[:CREATED]->(a1)
      AND (a1)-[:APPEARS_ON]-(:Track)
RETURN *
```

```
SemiApply(0)
|
+SemiApply(1)
|
+NodeByLabelScan(0)
|
+Filter(0)
|
+Expand(0)
|
+NodeByLabelScan(1)
|
+Filter(1)
|
+Expand(1)
|
+NodeByLabelScan(2)
```





- Need to run plan
- Plans are converted to nested iterators
- Result is obtained by pulling from the top iterator
- Room for improvement



Runtime Code Generation

Compiling Queries

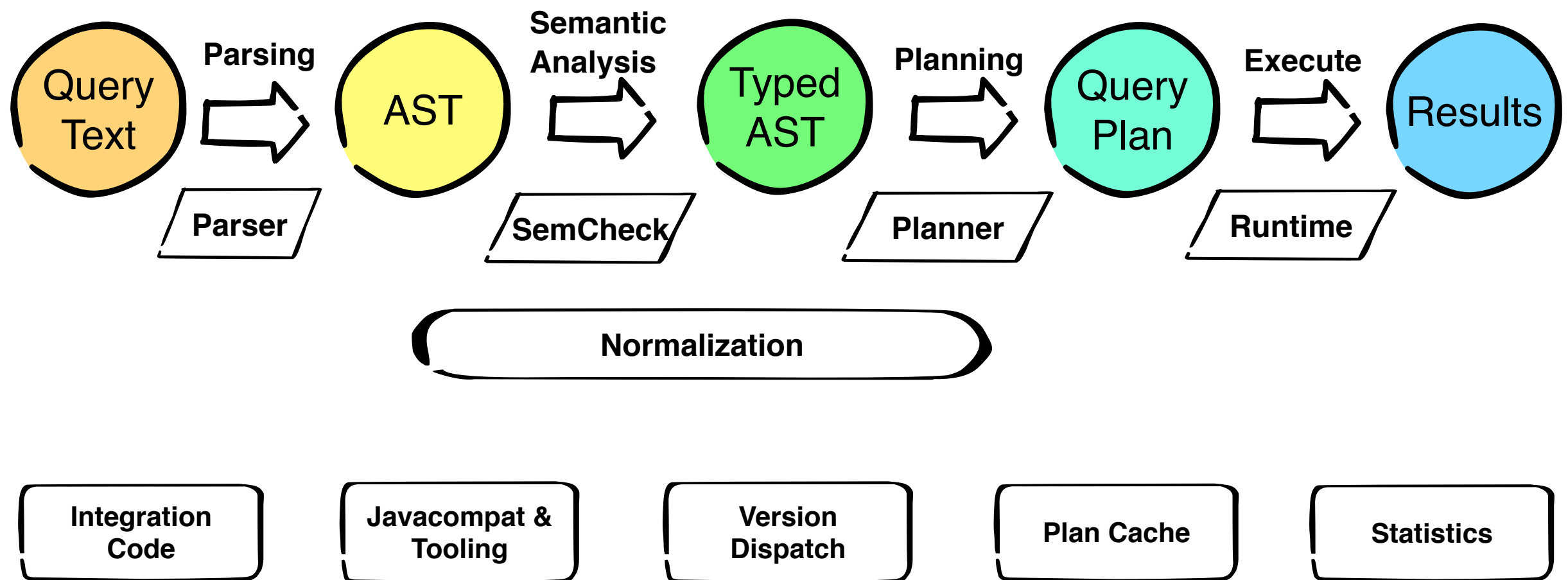
- Compilers >> Interpreters
- Cypher: Mix of dynamic and static types
- Static types: Ahead of time compilation
- Dynamic types: Runtime JIT, Tracing
- Mixing both: Gradual type systems

Code generation options on the JVM

- Generate strings and compile
- Write your own code generator
- Annotation Processing
- Use dynamic languages: Ruby, Clojure, JS
- Use static languages: scala, xtend
- No true staging language for the JVM

Truffle & Graal

- Truffle
 - Execution as evaluation of tree of nodes
 - Writer interpreter by writing new tree nodes
 - Runtime specialization for primitive types
- Extra speed up via Graal VM
 - Performance gains: x5 - x10

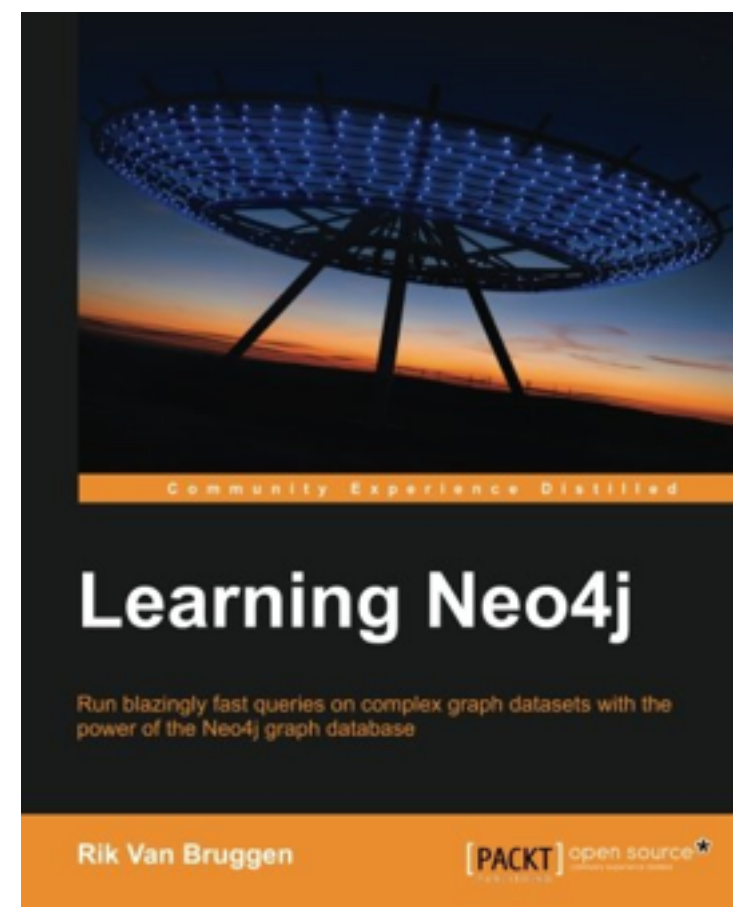
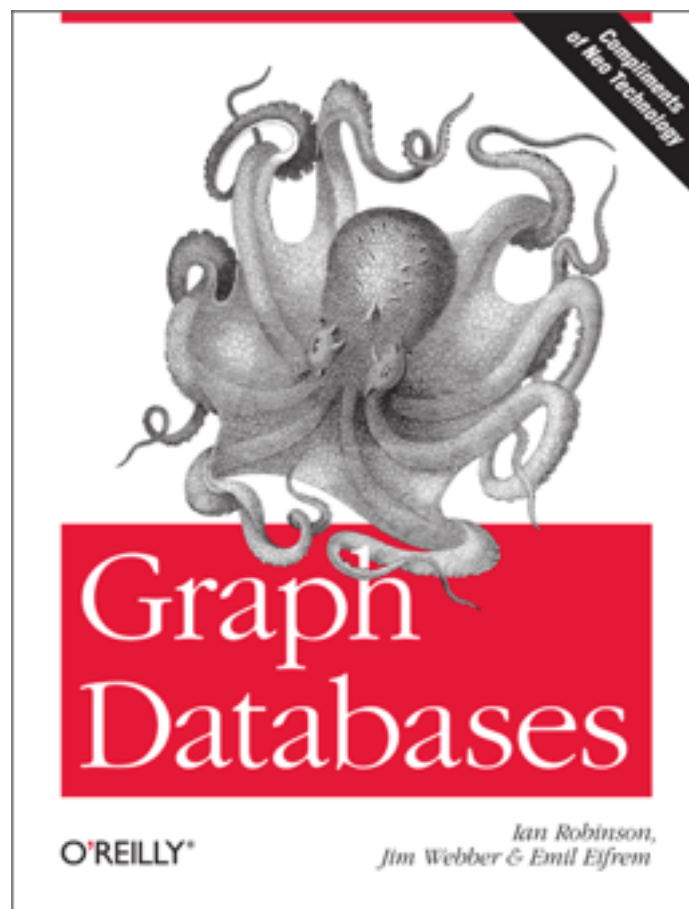


Neo4j Enterprise Graph Database Server

Takeaways

- Querying graphs with Cypher is fun and easy
- A cost based optimizer improves performance substantially
- Neo4j is a graph database

Learn More



<http://www.neotechnology.com>

Meet Neo4j



October 22
San Francisco

Meetups / User Groups



Neo4j meetups are worldwide. Make a connection or start a new group.

[Join a Meetup »](#)

<http://www.graphconnect.com>