Architecture

neo4j in short

Feature	neo4j
Model	Graph-based
Query language	Supported, Cypher
Reference scenarios	transactional (read intensive) & analytical
Partitioning	Difficult (application-level)
Indexes	on properties (simple and composite), full-text
Replication	Master-slave, single-leader
Consistency	Strong
Availability	Load balancing among read replicas
Fault tolerance	By re-electing a master in case it goes down
Transactions	ACID (mantain consistency over multiple nodes and edges)
CAP theorem	CA
Distributed by	Neo4j Inc.

Indexes and Query Tuning

- Indexes can be added to improve search performance

  CREATE INDEX [index_name]
  FOR (n:LabelName)
  ON (n.propertyName)

  - Different indexes, recommended BTREE

- Composite (multiproperty) indexes can be defined as well
- Full-text indexes can be defined as well (Apache Lucene)

- The use of indexes by the query optimizer can be checked by looking at the query plan (EXPLAIN) or suggested by USING

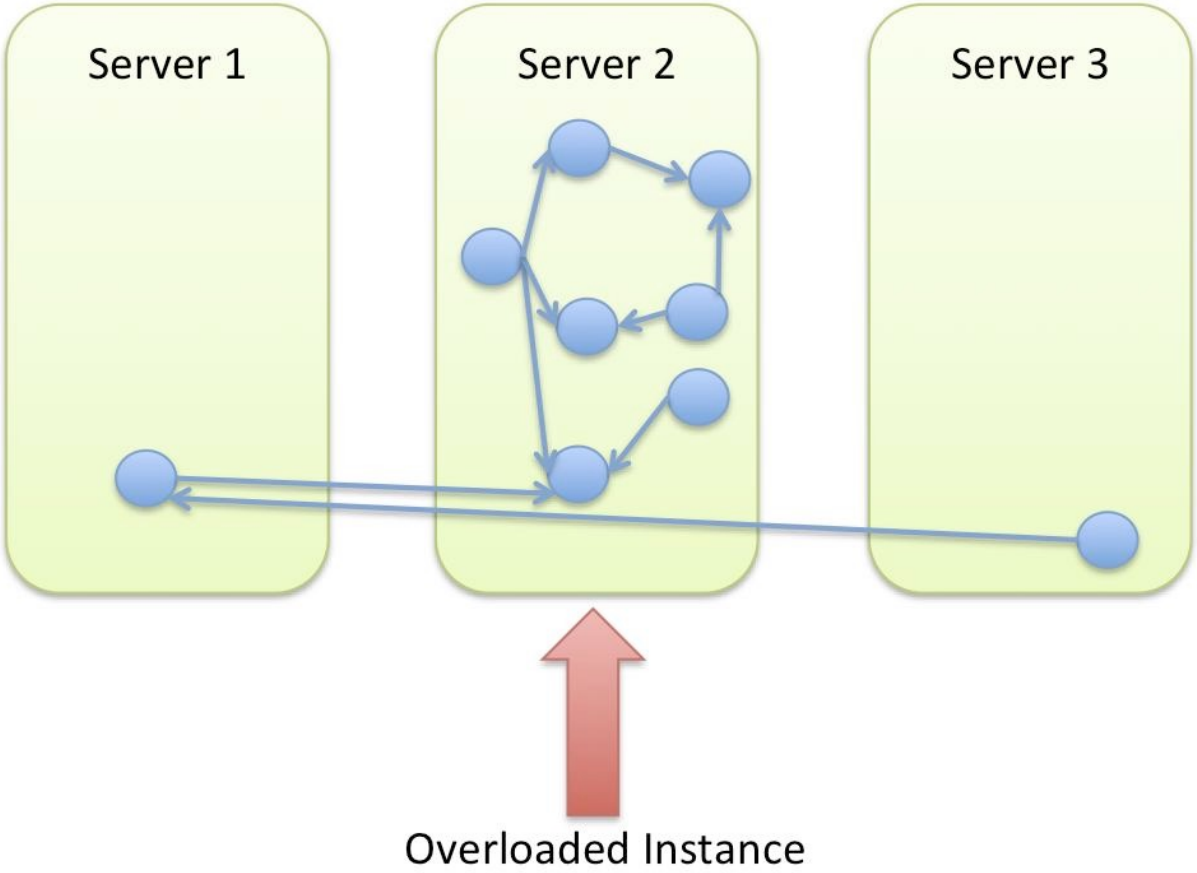
https://neo4j.com/docs/cypher-manual/current/administration/indexes-for-search-performance/

Graph Partitioning and Federated Access

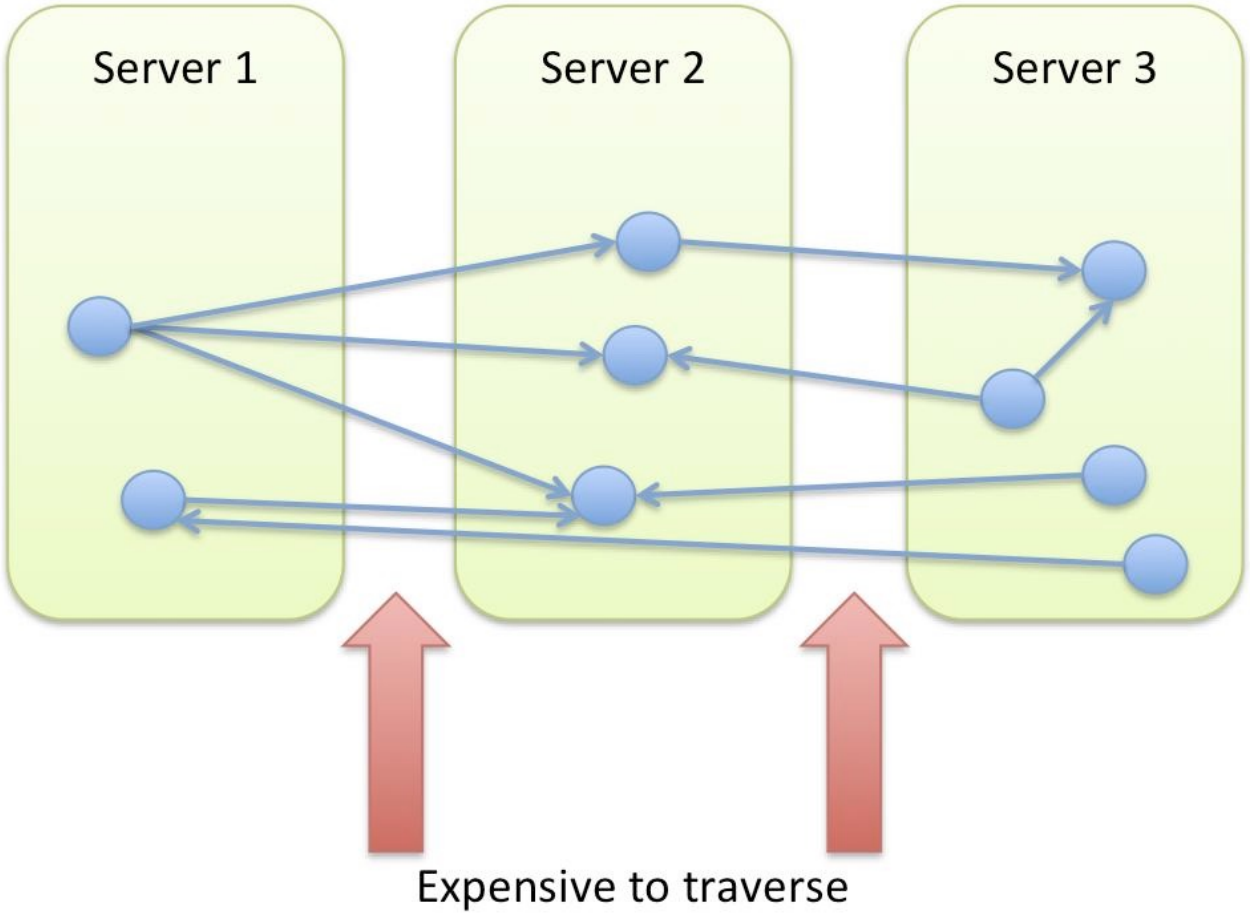
Paritioning

- Partitioning is difficult
- Storing related nodes on the same server is better for graph traversal
- Traversing a graph when the nodes are on different machines is not good for performance
- But any node can be related to any other node …

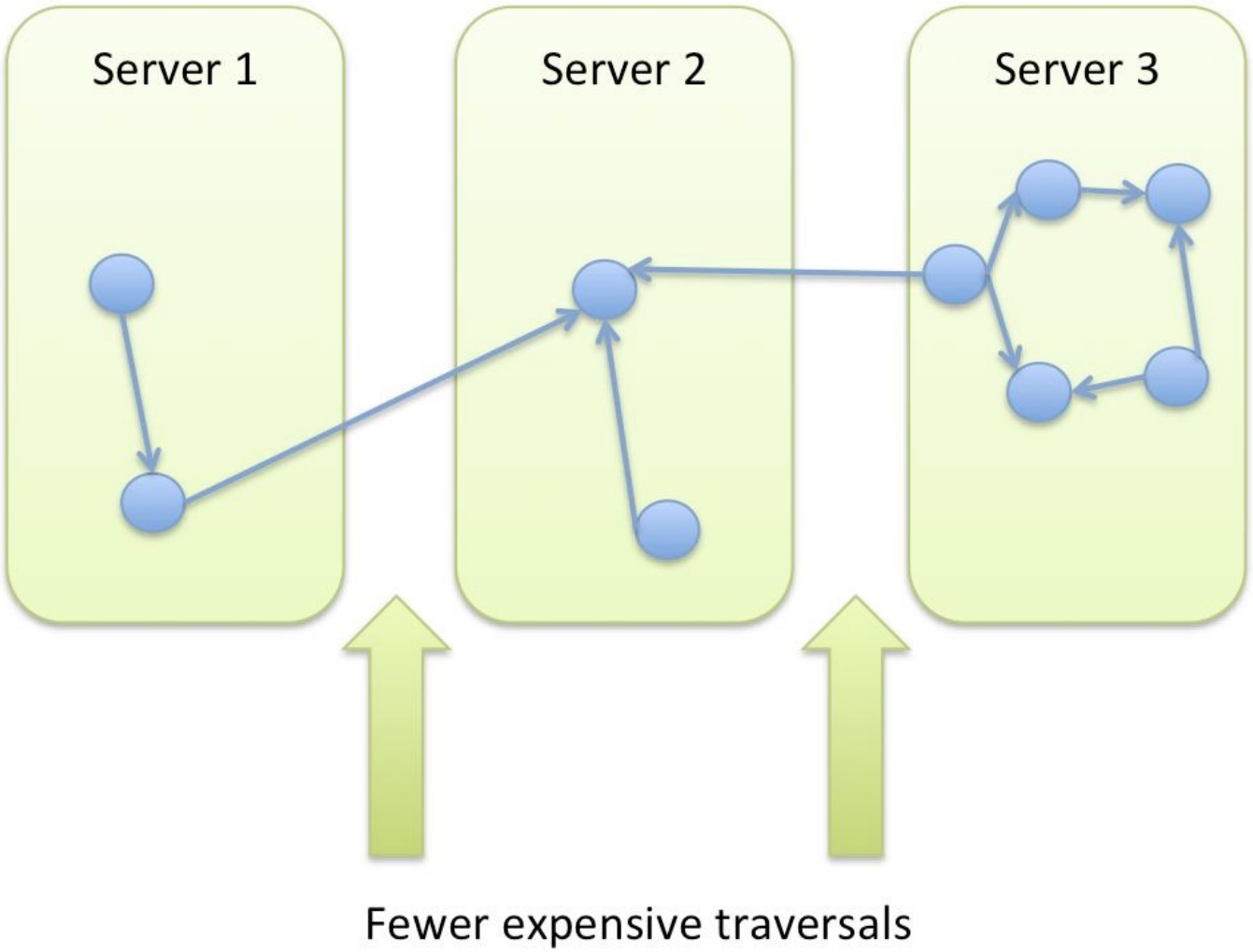
Partitioning - “Black Hole” server



Partitioning - Chatty Network



Paritioning - Minimal Point Cut

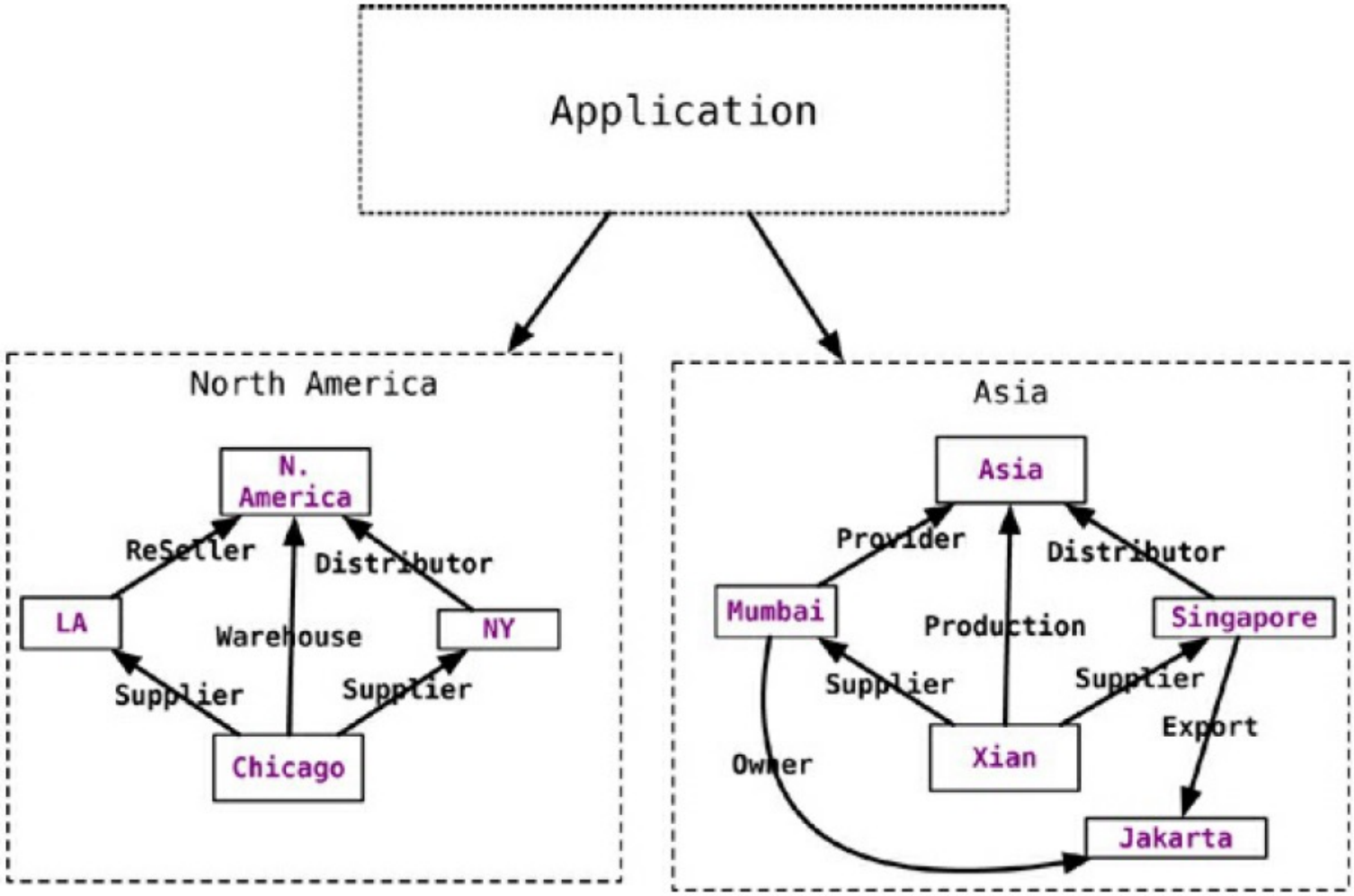


Application-level Partitioning

Partition the data from the application side using domain-specific knowledge

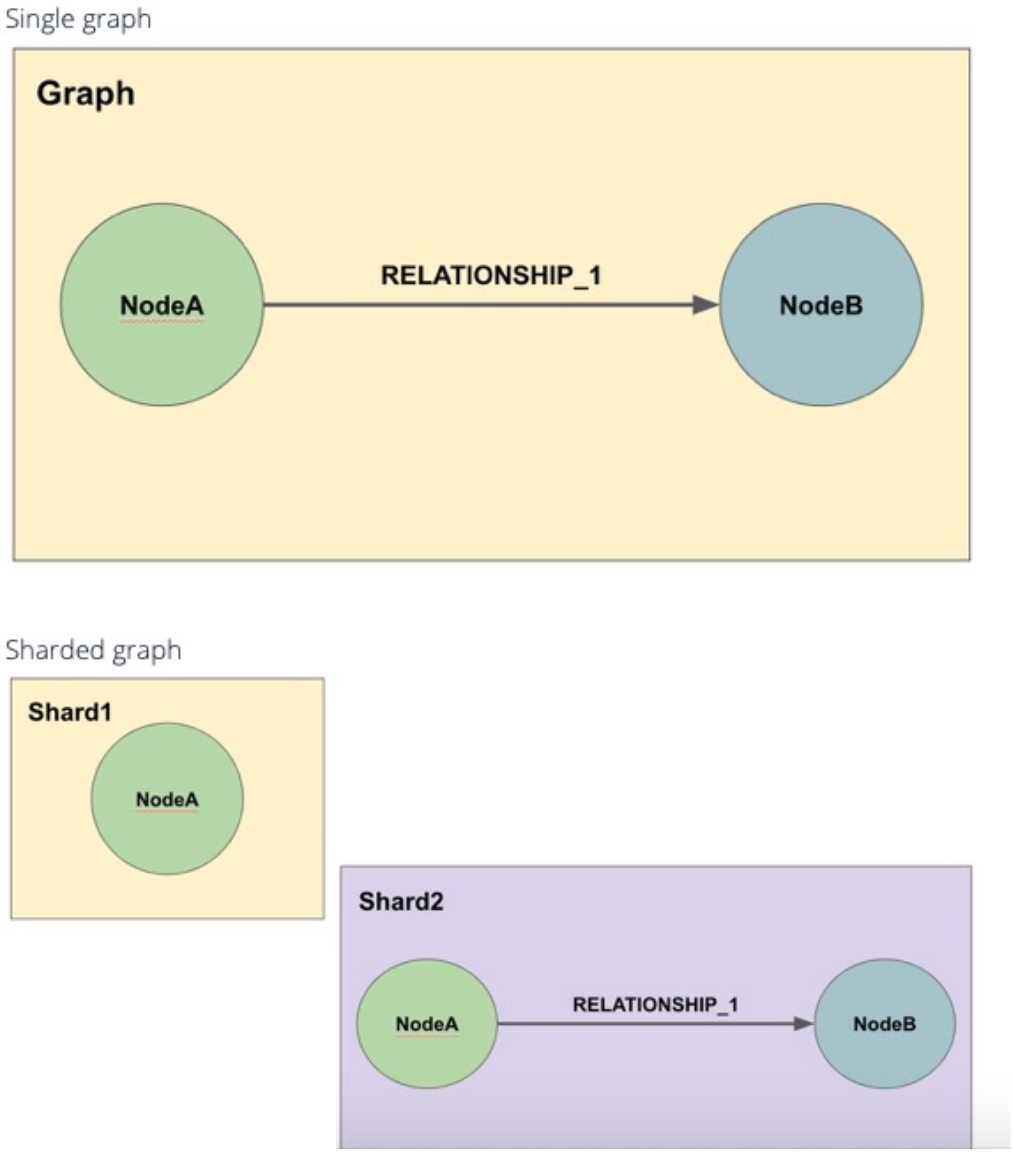
- For example, nodes that relate to the North America can be created on one server while the nodes that relate to Asia on another
- This application-level partioning needs to understand that nodes are stored on physically different databases

Application-Level Partitioning



Partitioning Data with Neo4j Fabric

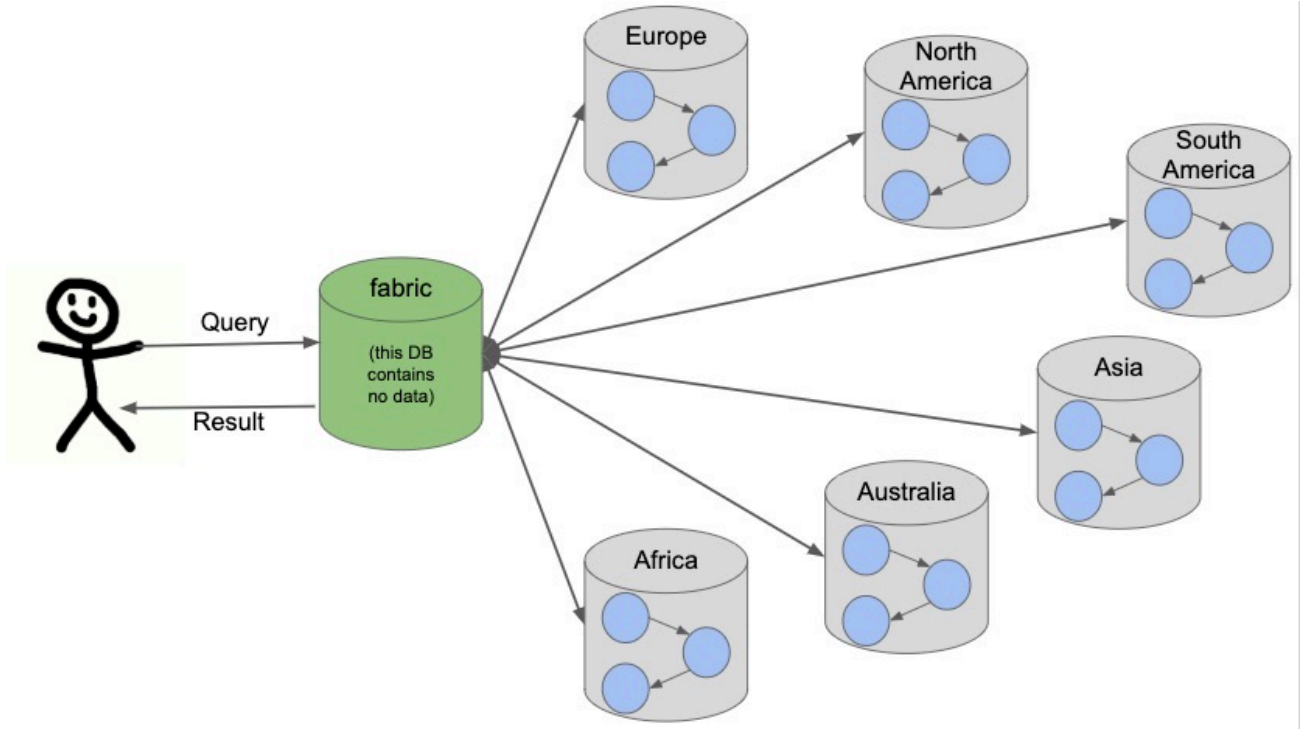
- Enable users to break a larger graph down into individual, smaller graphs and to store them in separate databases
- For highly-connected graphs, some **redundancy** to maintain the relationships between entities



https://neo4j.com/developer/neo4j-fabric-sharding/

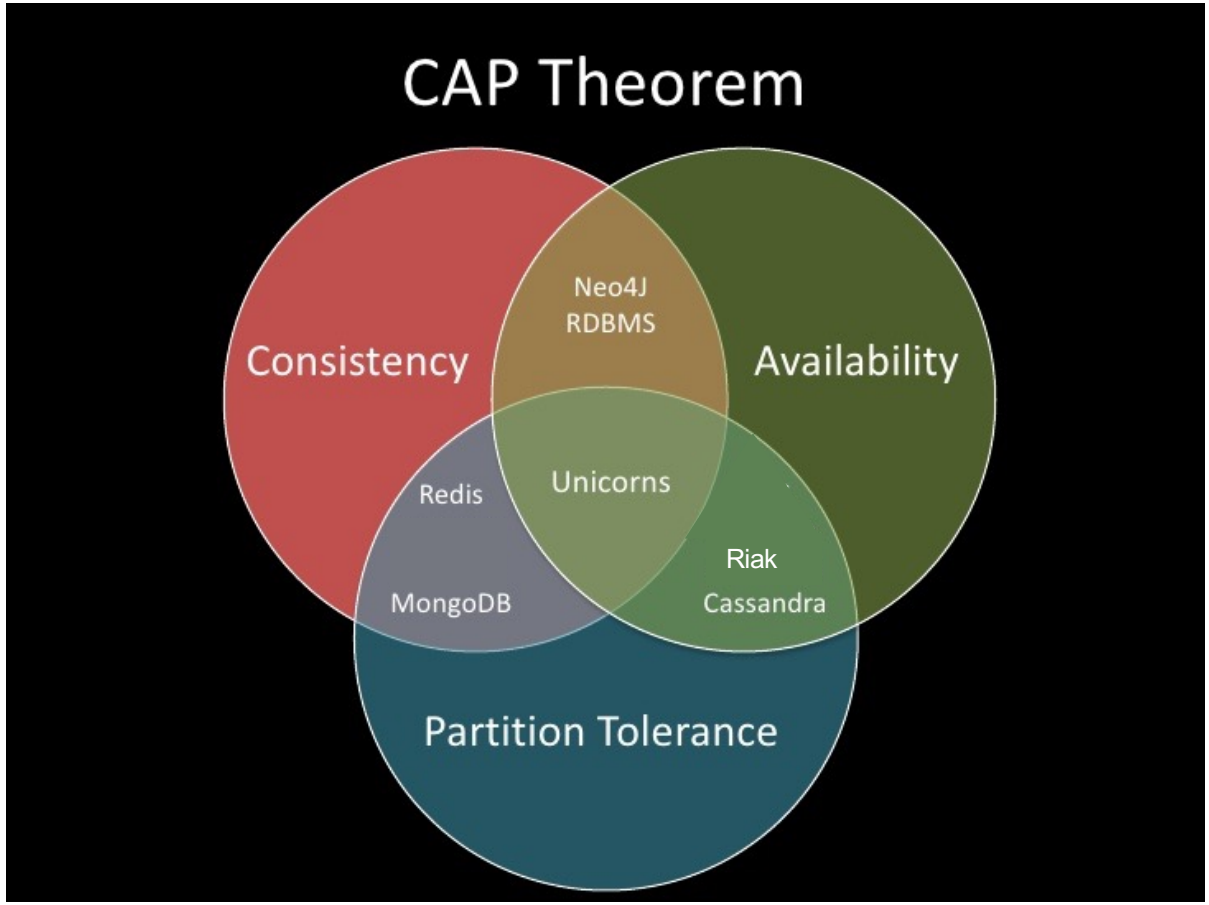
Partitioning with Neo4j Fabric

- The Fabric database is actually a virtual database
  - it cannot store data, but acts as the entrypoint into the rest of the graphs
  - like a proxy server that handles requests and connection information
  - it helps distribute load and sends requests to the appropriate endpoint



Neo4j on a cluster?
Scalability and availability

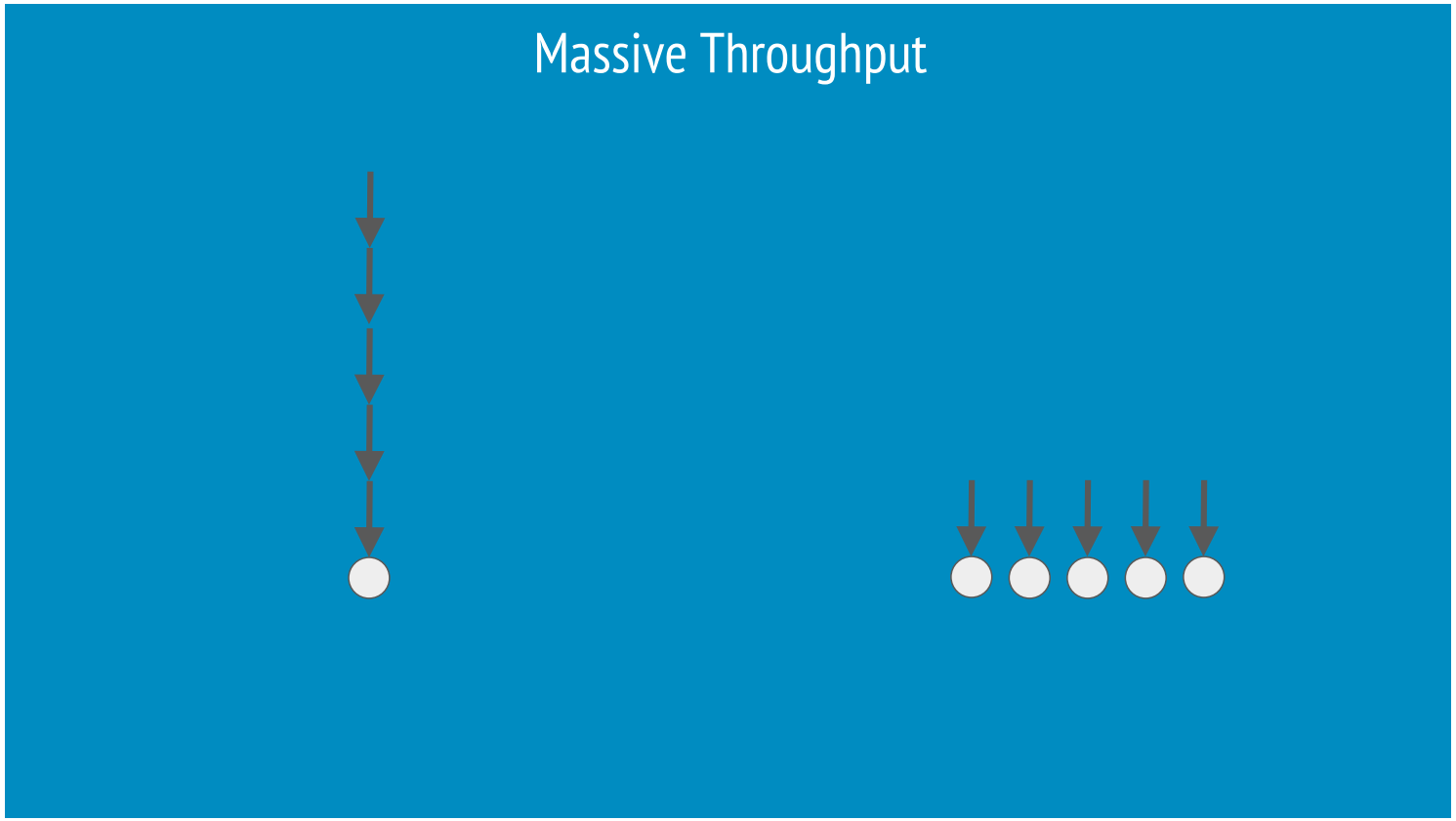
Consistency & Availability



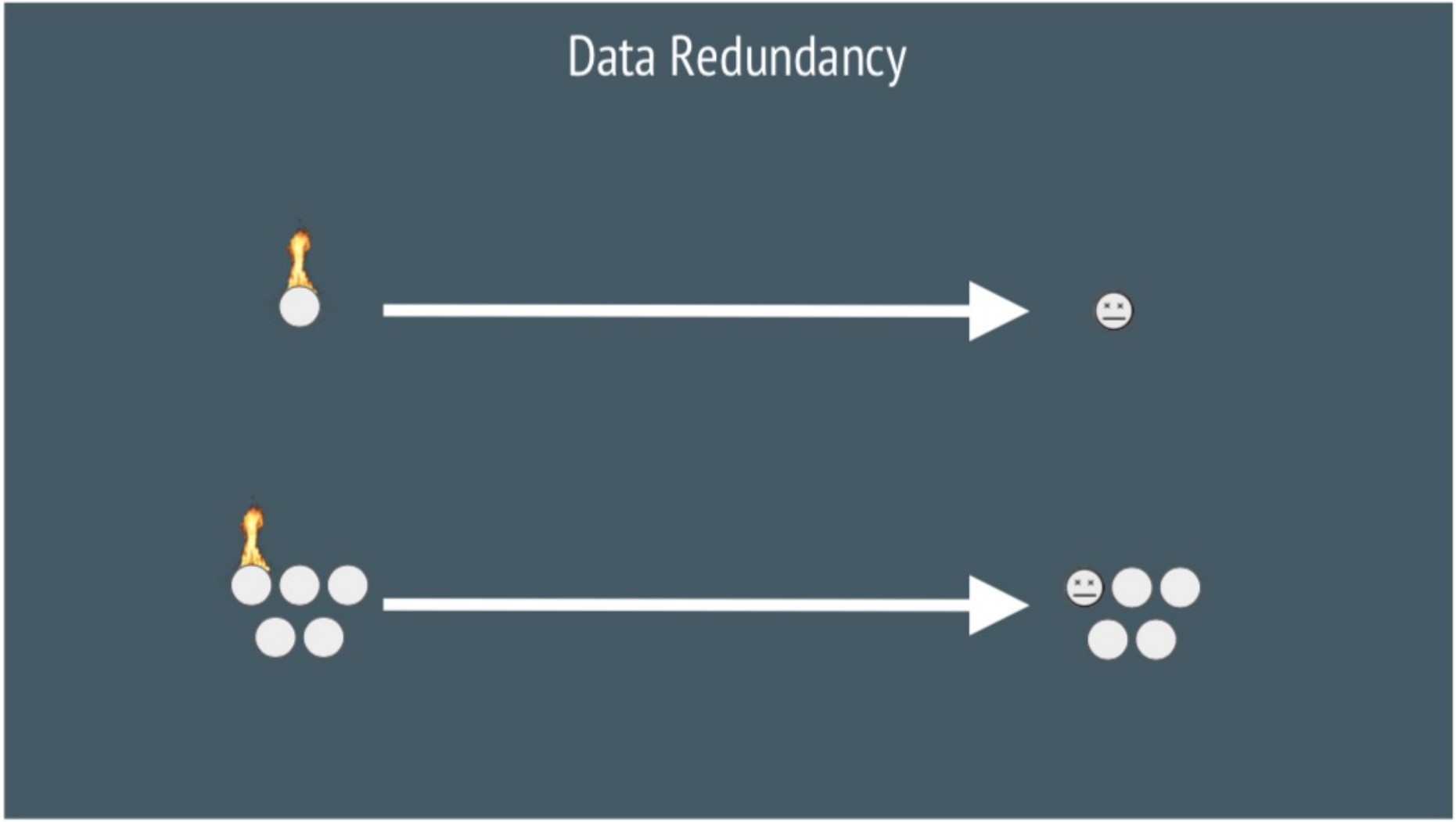
Master-slave replication model

ACID-compliant transactions

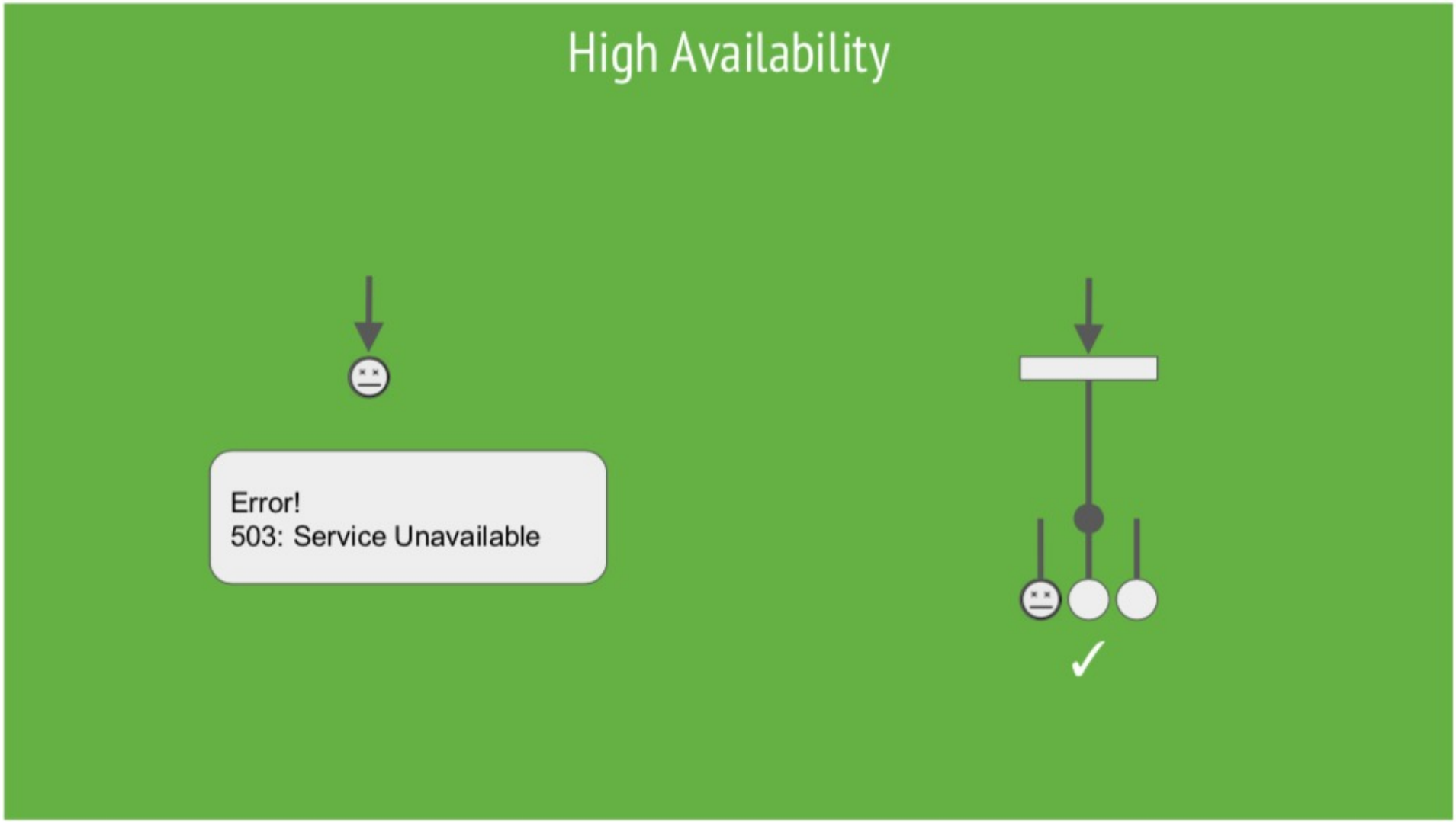
Even if we cannot partition, can graph dbs exploit running on a cluster?



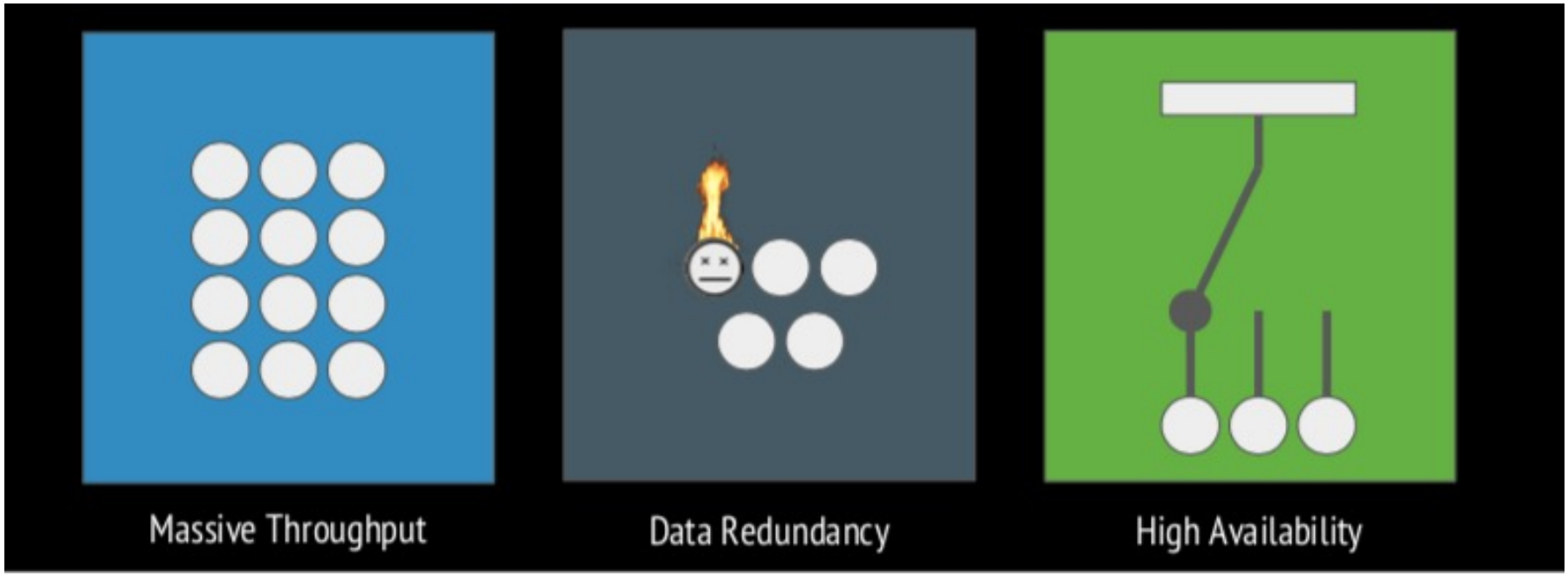
Even if we cannot partition,
can graph dbs exploit running on a cluster?



Even if we cannot partition,
can graph dbs exploit running on a cluster?



Even if we cannot partition,
can graph dbs exploit running on a cluster?



Availability & Scaling

- Master-slave replication
  - Several slave databases can be configured to be exact replicas of a single master database
- Speed-up of read operations
  - A horizontally scaling read-mostly architecture
  - Enables to handle more read load than a single node
- Fault-tolerance
  - In case a node becomes unavailable
- Transactions are still atomic, consistent and durable, but eventually propagated to the slaves

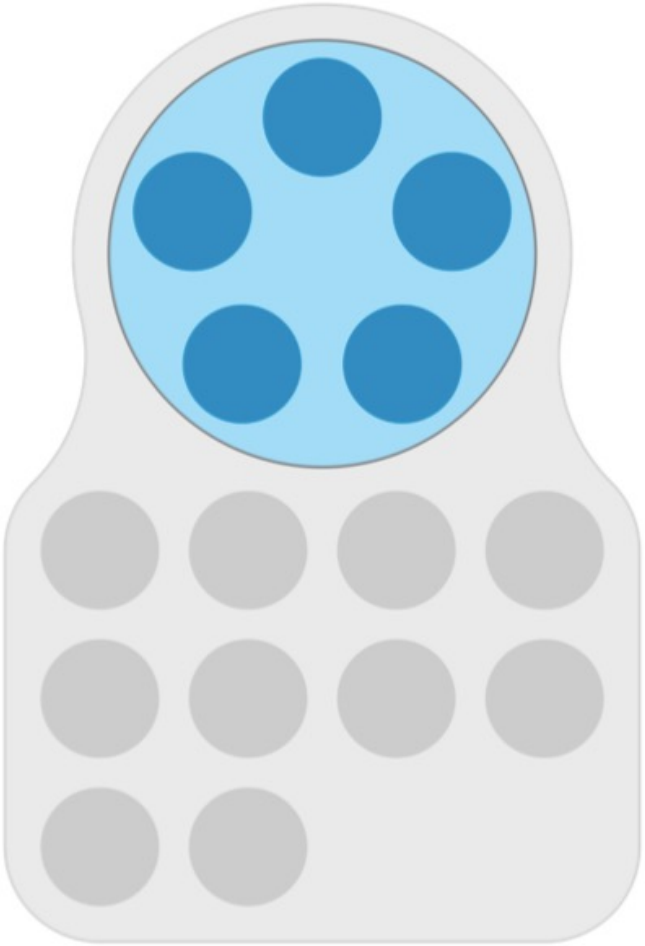
Consistency

- Consistency ensured through transactions
- All Cypher statements are explicitly run within a transaction
  - (full ACID-compliance holds for most graph databases as well)
- When running in a cluster
  - a write to the master is eventually synchronized to the slaves
  - slaves are always available for read
  - writes to slaves maybe allowed
    - they are immediately synchronized to the master and result in success only after having been committed at the master
    - other slaves will not be synchronized immediately (they will have to wait for the data to propagate from the master)

Availability

- High availability can be achieved by providing for replicated slaves
- Slaves can also handle writes:
  - When they are written to, they synchronize the write to the current master, and the write is committed first at the master and then at the slave
  - Other slaves will eventually get the update
- Keeping track of the last transaction IDs persisted on each slave node and the current master node
  - Once a server starts up, it finds out which server is the master
  - If the server is the first one to join the cluster, it becomes the master
  - When a master goes down, the cluster elects a master from the available nodes

Consensus Commit: Core

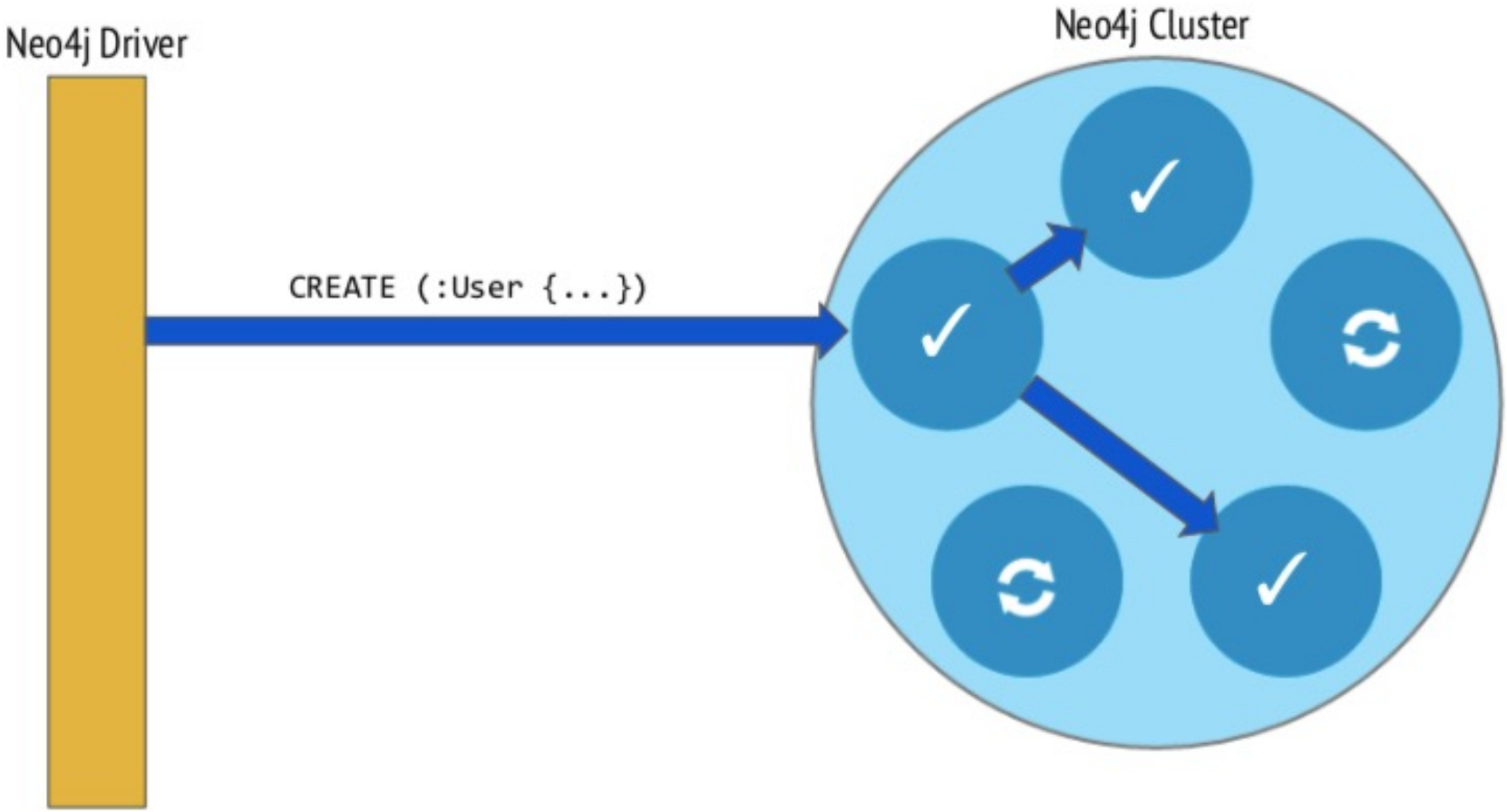


Core

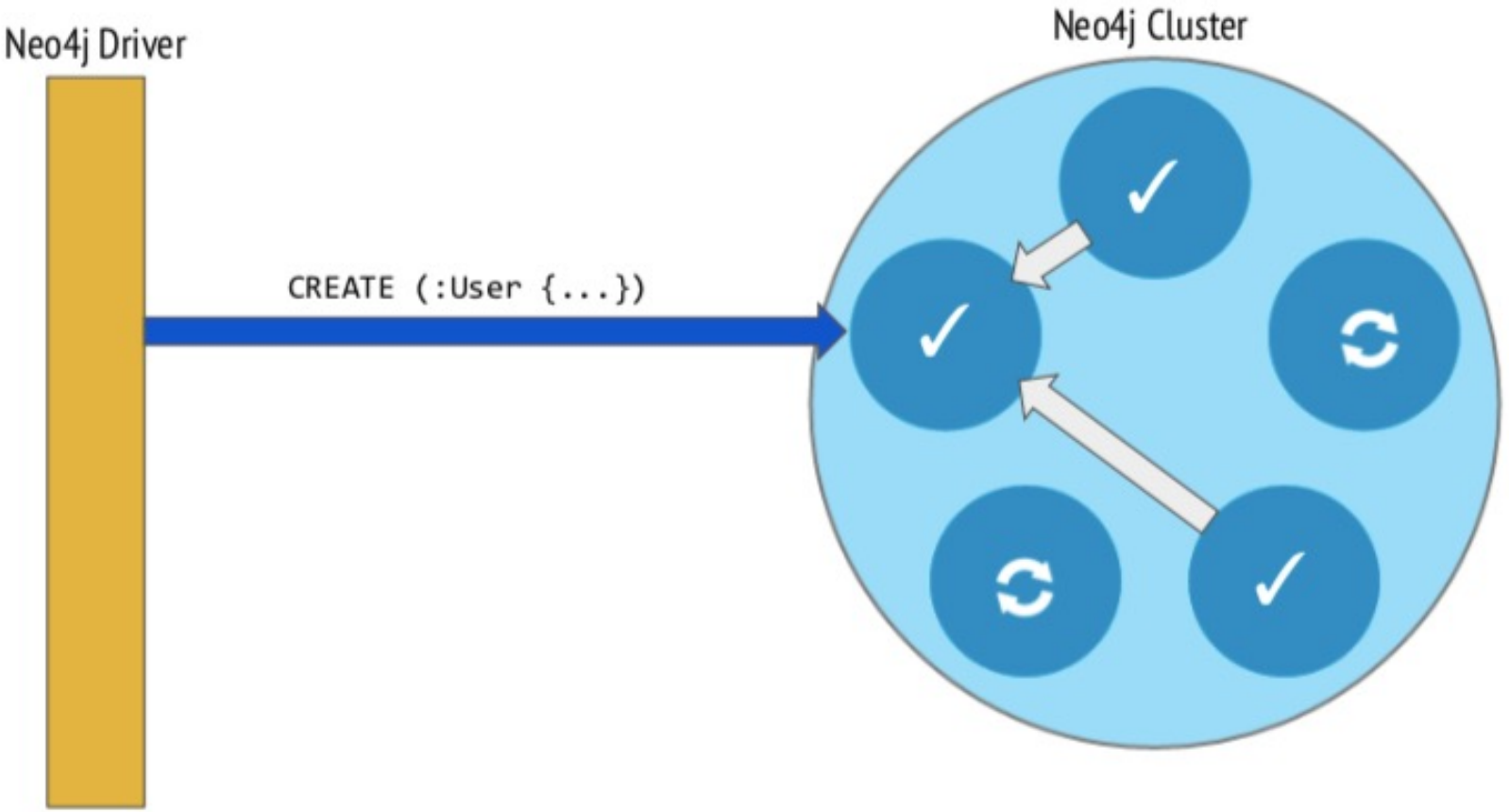
- Small group of Neo4j databases
- Fault-tolerant Consensus Commit
- Responsible for data safety

Raft consensus protocol

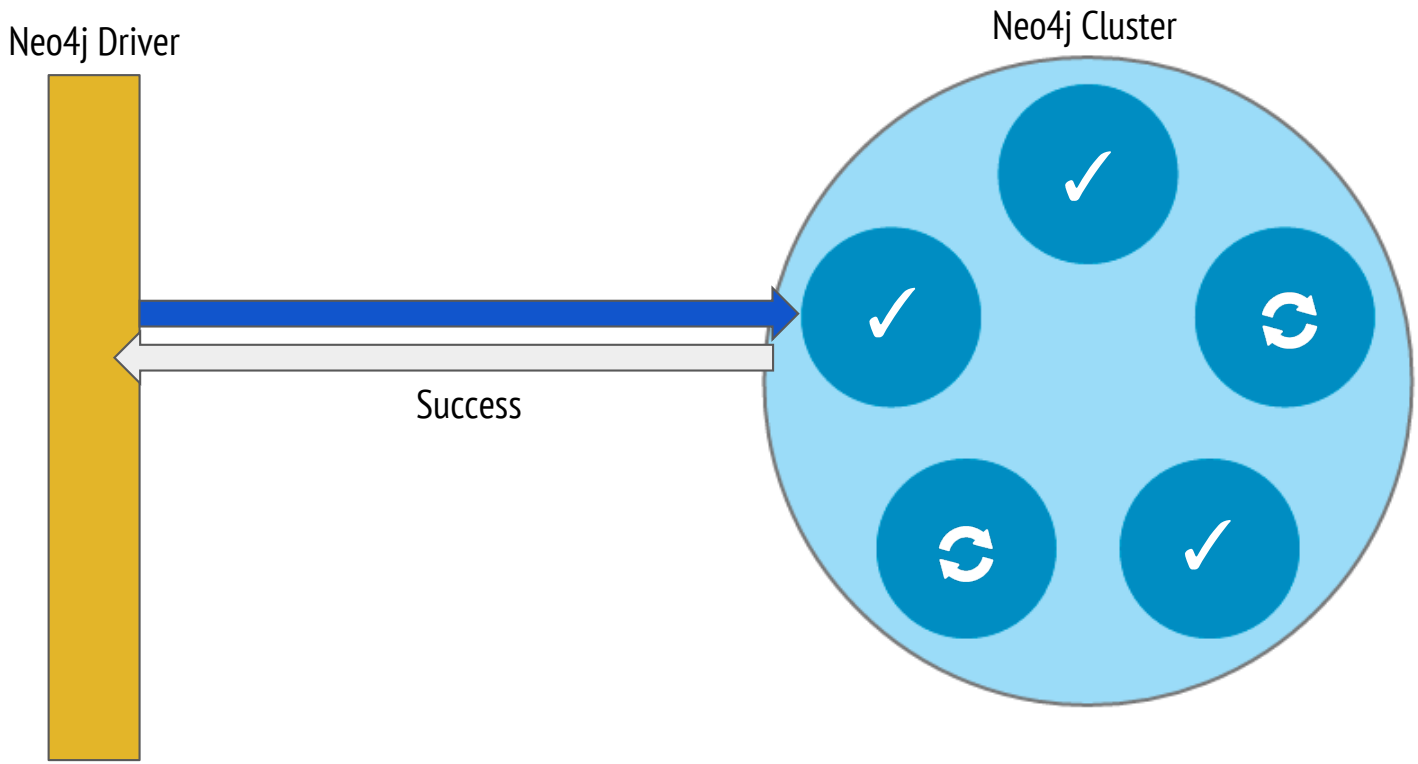
Writing to Core



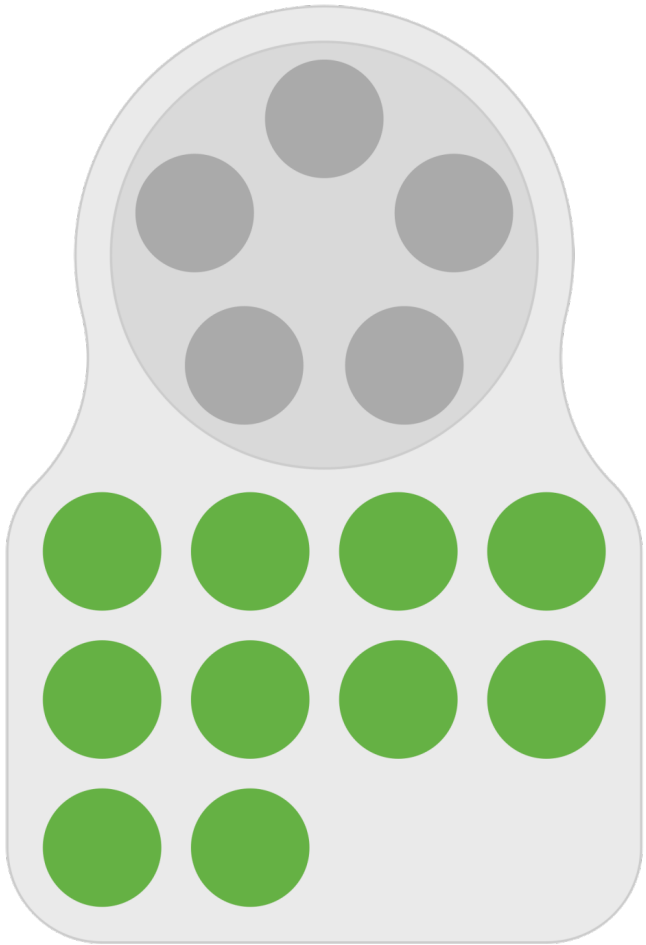
Writing to Core



Writing to Core



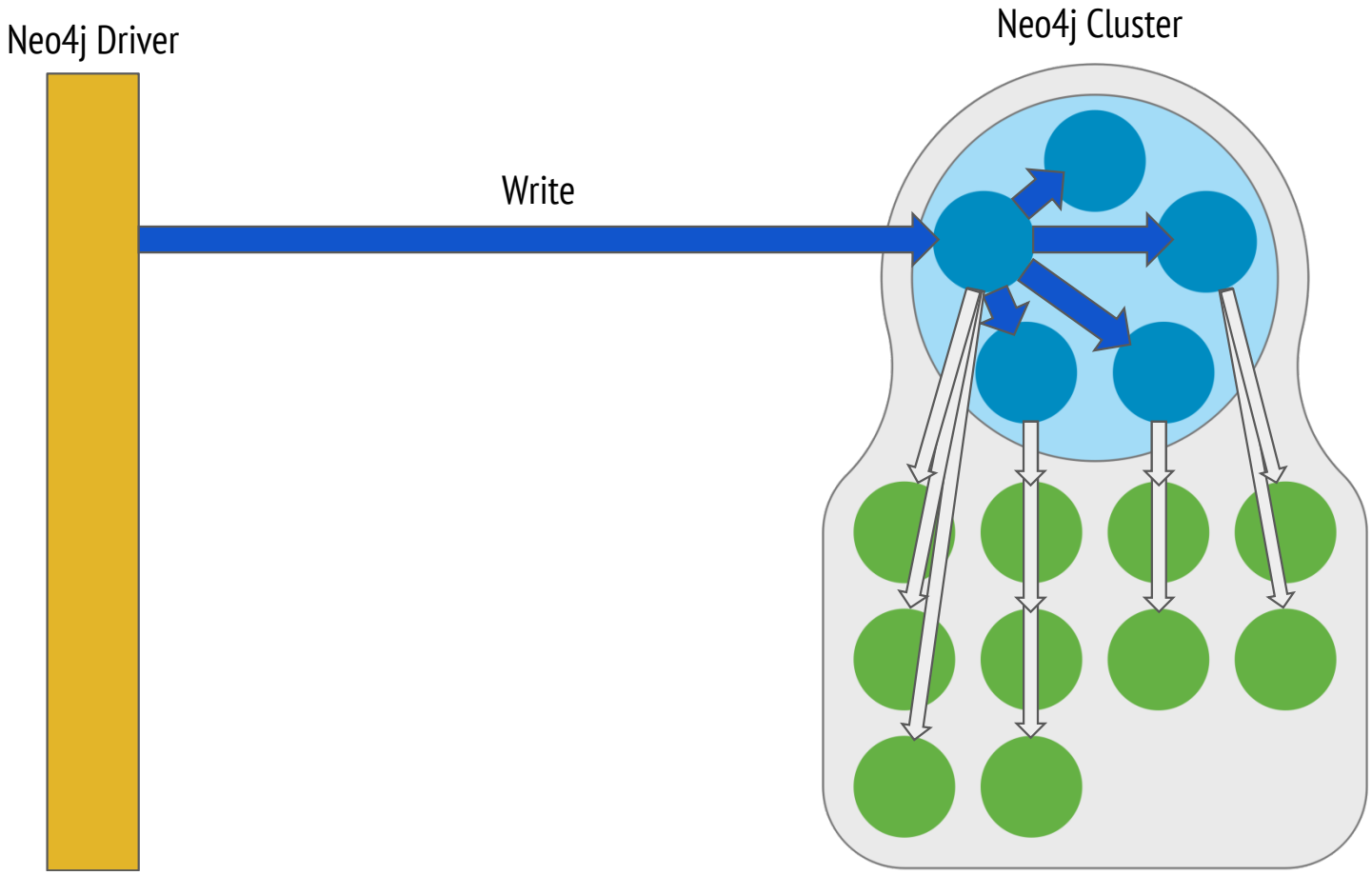
Read Replicas



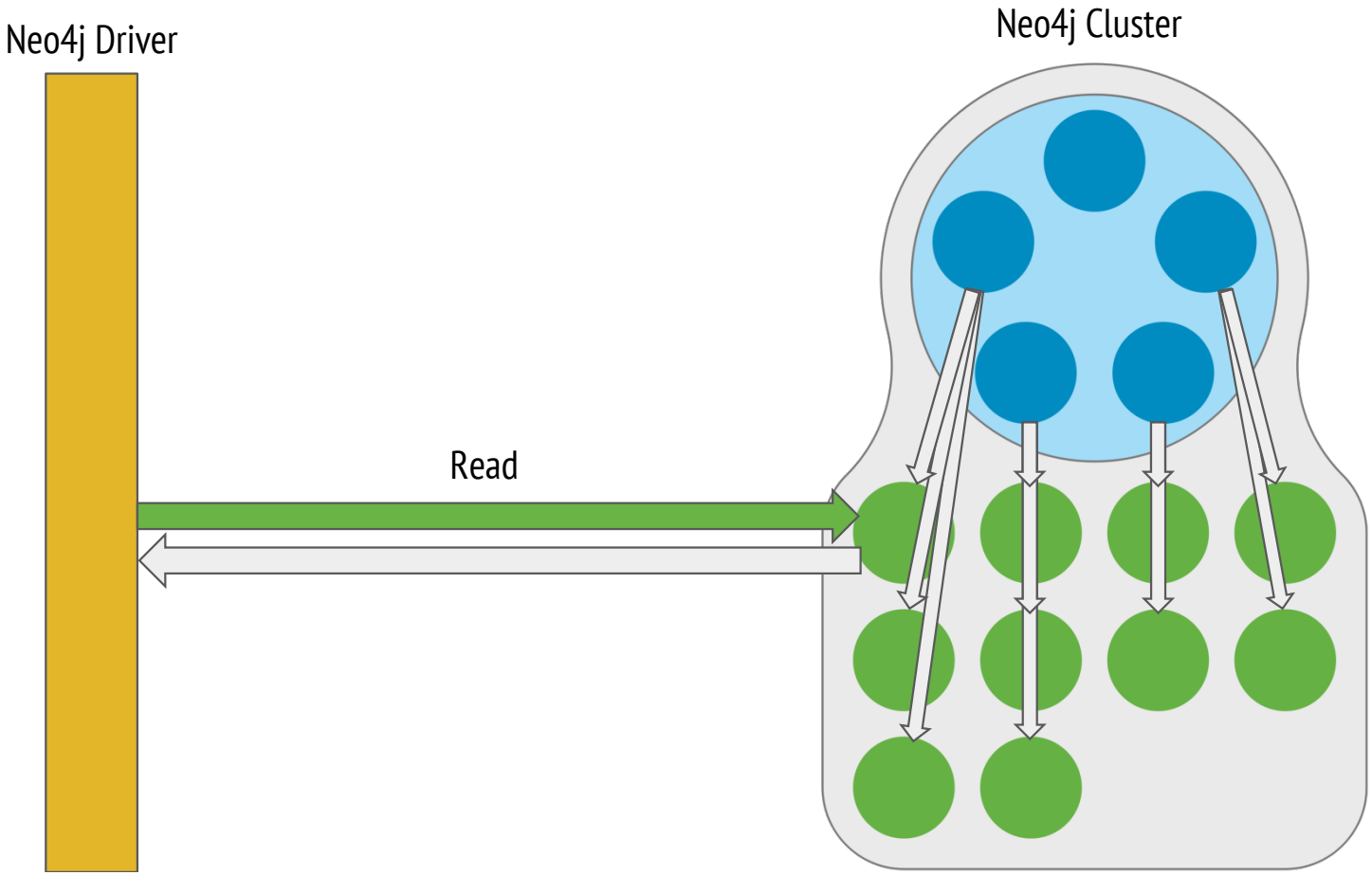
Read Replicas

- For massive query throughput
- Read-only replicas
- Not involved in Consensus Commit
- Disposable, suitable for auto-scaling

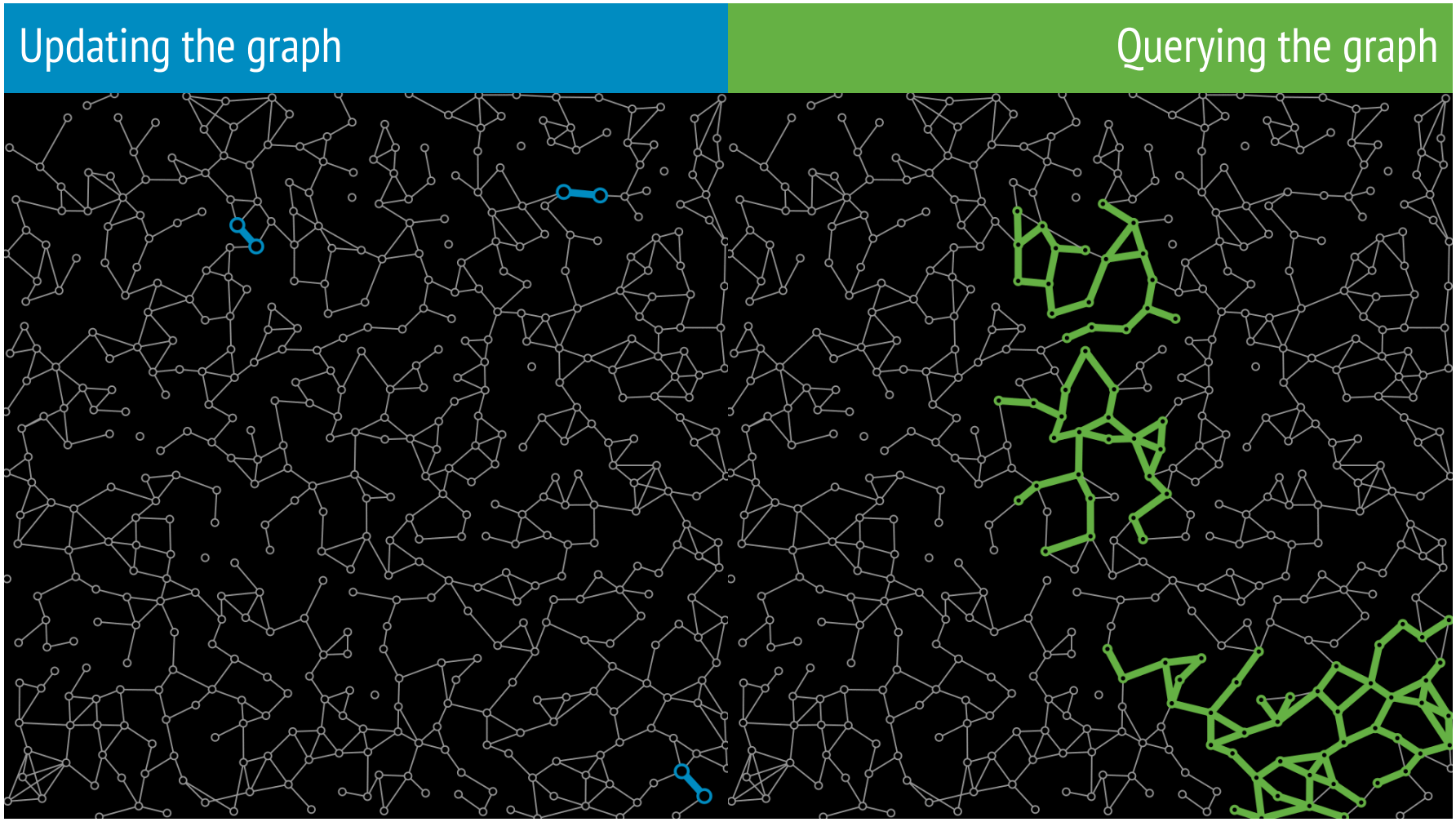
Propagating Updates to Read Replicas



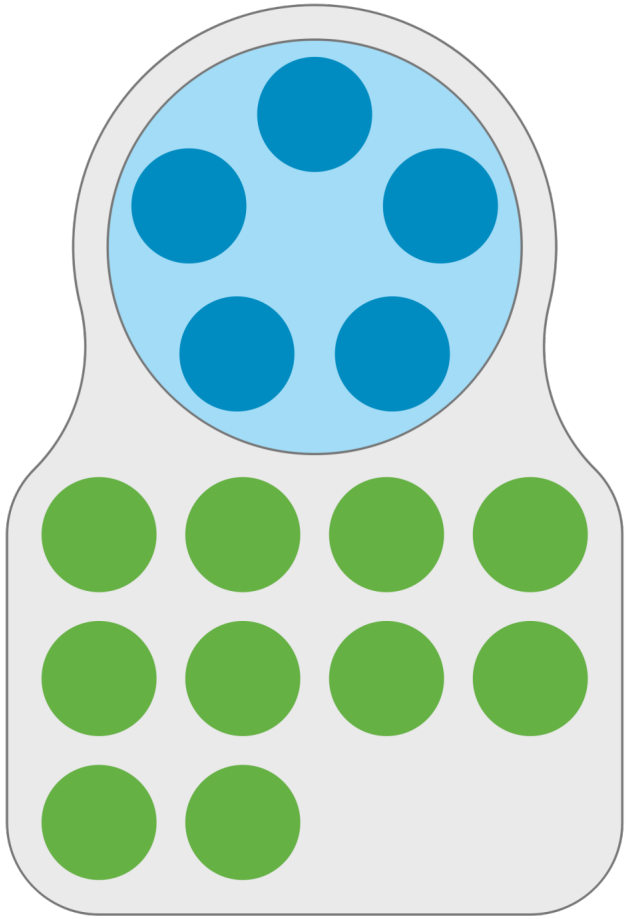
Reading from Read Replicas



Updating vs Reading



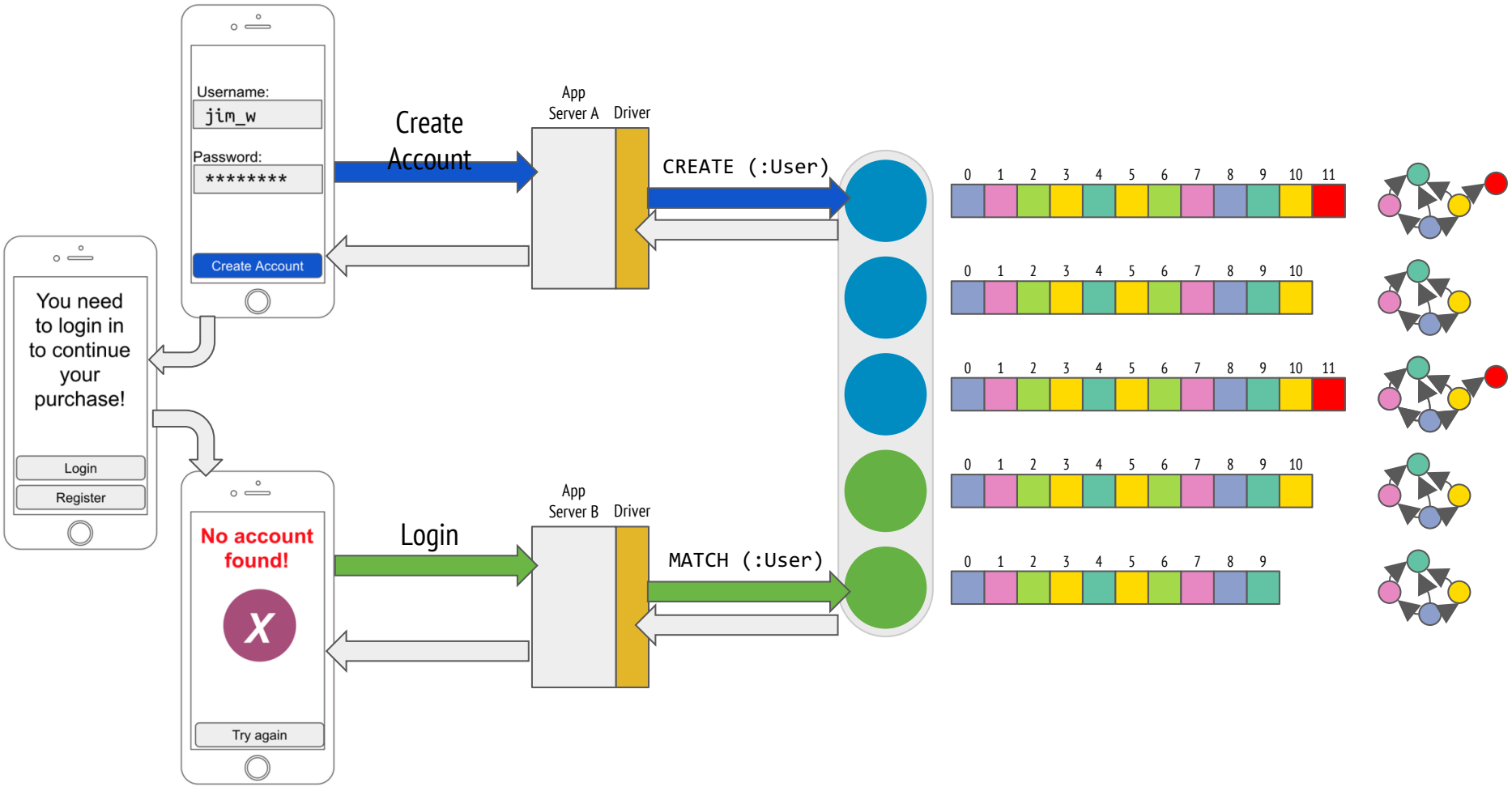
Updating vs Reading



Core Updating the graph

Read Replicas Queries, analysis, reporting

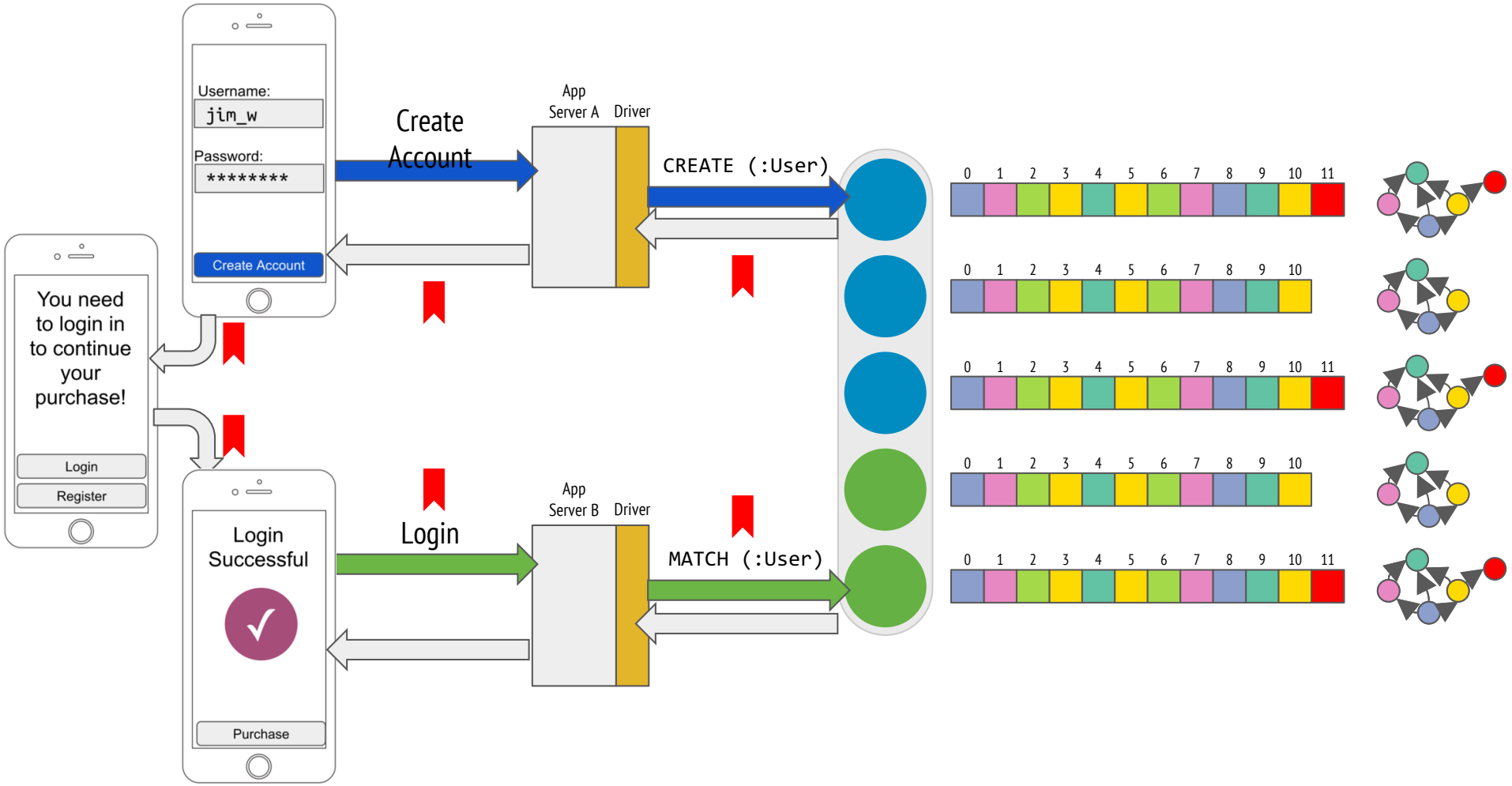
From Eventual to Causal Consistency



From Eventual Consistency to Causal Consistency

- On executing a transaction, the client can ask for a bookmark which it then presents as a parameter to subsequent transactions
- This way, the cluster can ensure that only servers which have processed the client's bookmarked transaction will run its next transaction
- This provides a *causal chain* which ensures correct read-after-write semantics from the client's point of view

From Eventual to Causal Consistency - Bookmarks



when invoked, a client application is guaranteed to read at least its own writes

Feature	In Neo4j (EE)
Partitioning	Difficult Application-level
Replication	Core nodes & read replicas
Consistency	Causal consistency (read your own writes)
Availability	Load balancing among read replicas for read (core nodes for writes)
Fault tolerance	Consensus commit among core nodes (Raft protocol)
Transactions	ACID (mantain consistency over multiple nodes and edges)
CAP theorem	CA

Use Cases

Graph DBs: Suitable Use Cases

- Connected Data
  - Social networks
  - Any link-rich domain is well suited for graph databases
- Routing, Dispatch, and Location-Based Services
  - Node = location or address that has a delivery
  - Graph = nodes where a delivery has to be made
  - Relationships = distance
- Recommendation Engines
  - “your friends also bought this product”
  - “when buying this item, these others are usually bought”

Graph DBs: When Not to Use

- If we want to update all or a subset of entities
  - Changing a property on many nodes is not straightforward
    - e.g., analytics solution where all entities may need to be updated with a changed property
- Some graph databases may be unable to handle lots of data
  - Distribution of a graph is difficult

References

- Ian Robinson, Jim Webber & Emil Eifrem, Graph Databases, New Opportunities for Connected Data, 2nd Edition, O'Reilly, 2015
- Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 192 p.
- Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications
- Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, Jonas Partner, Neo4J in Action, Manning, 2015
- http://neo4j.com/docs

Credits

- Credits for the slides:
  - Antonio Maccioni, Università Roma Tre
  - Kevin Swingler, University of Stirling
  - David Novak, Masaryk University, Brno
  - Irena Holubova, Charles University Praha
  - Jim Webber, Kevin Van Gundy, neo4j