CISC 7610 Lecture 4
Approaches to multimedia databases

Topics:
Graph databases
Neo4j syntax and examples
Document databases
MongoDB syntax and examples
Column databases

NoSQL architectures: different tradeoffs for different workloads

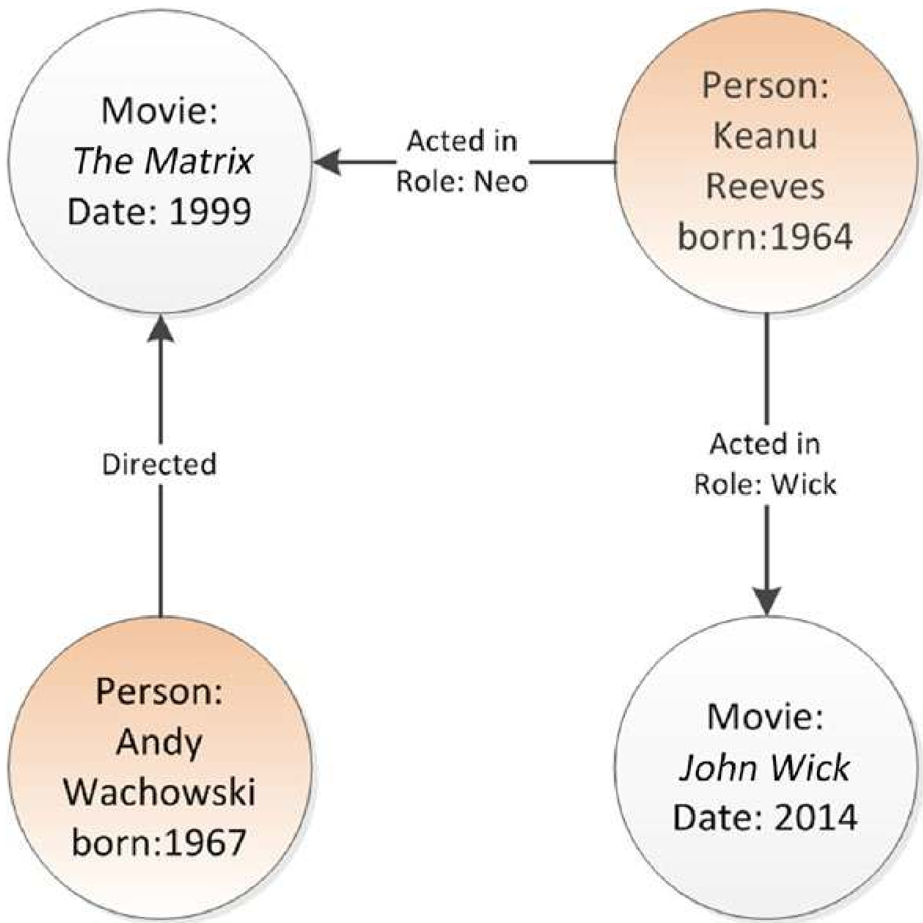
- Already seen: Generation 1
  - Hadoop for batch processing on commodity hardware
  - Key-value stores for distributed non-transactional processing
- This lecture: Generation 2
  - Document databases for better fit with object-oriented code
- This lecture also: Generation 3
  - Graph databases for modeling relationships between things
  - Column stores for efficient analytics

Graph databases

Graph databases

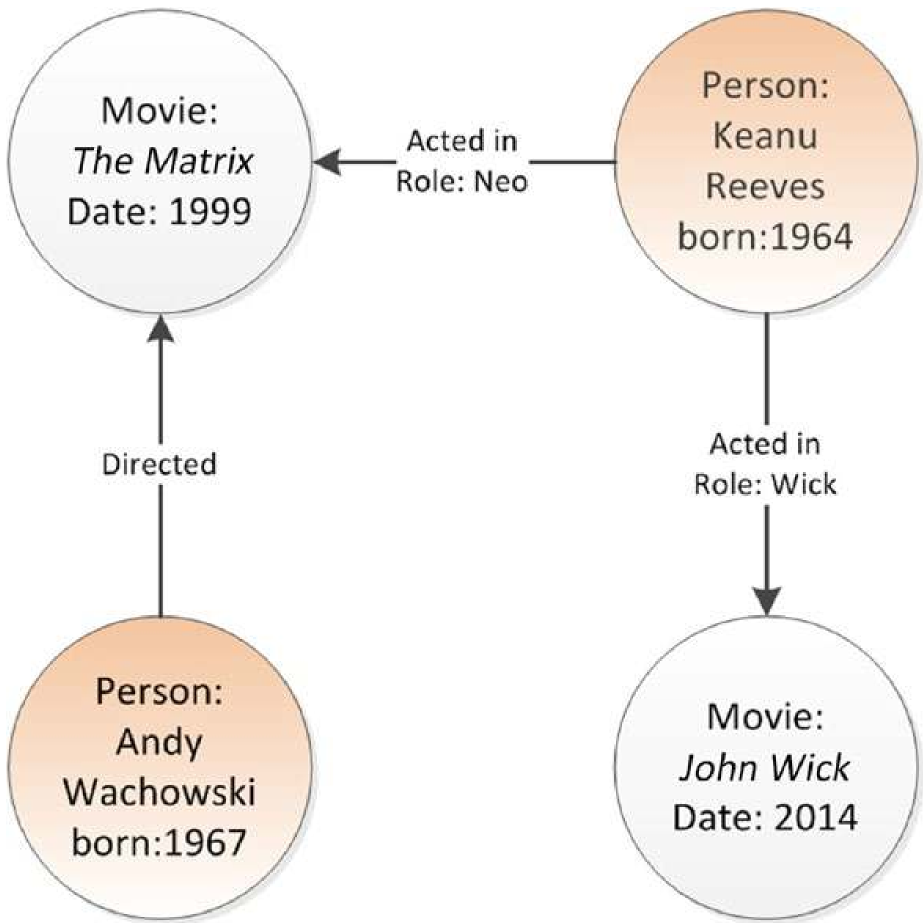
- Most databases store **information about things**: key-value stores, document databases, RDBMS
- Graph databases put the **relationships between things** on equal footing with the things themselves
- Examples: social networks, medical models, energy networks, access-control systems, etc.
- Can be modeling in RDBMSs using foreign keys and self-joins
  - Generally hits performance issues when working with very large graphs
  - SQL lacks an expressive syntax to work with graph data
- Key-value stores and document databases lack joins, would treat a graph as one document
- For multimedia, store metadata in graph, data in key-value store

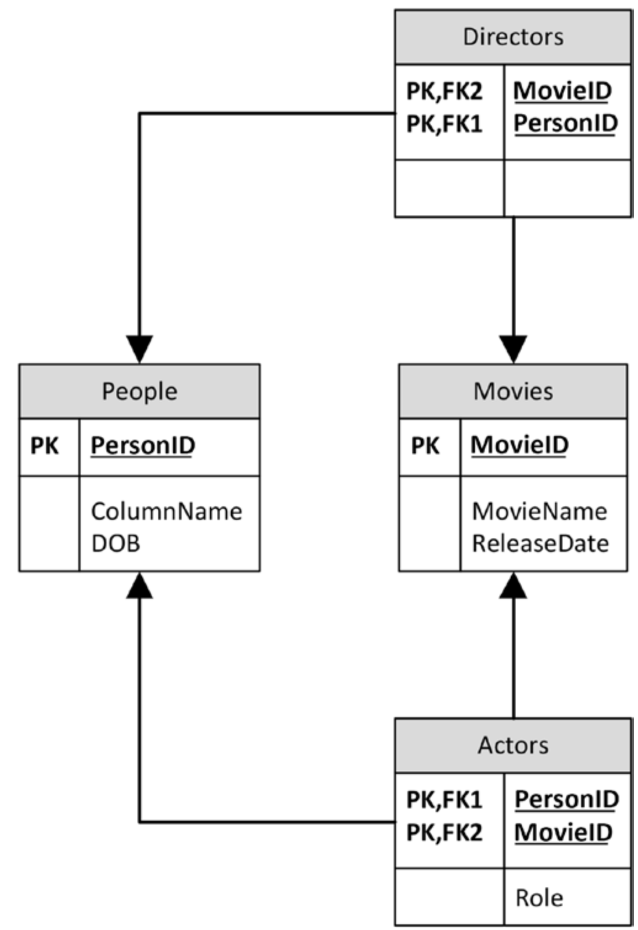
Example graph



- Graphs have
- Vertices (AKA "nodes")
- Edges (AKA "relationships")
- Both can have "properties"

Example graph



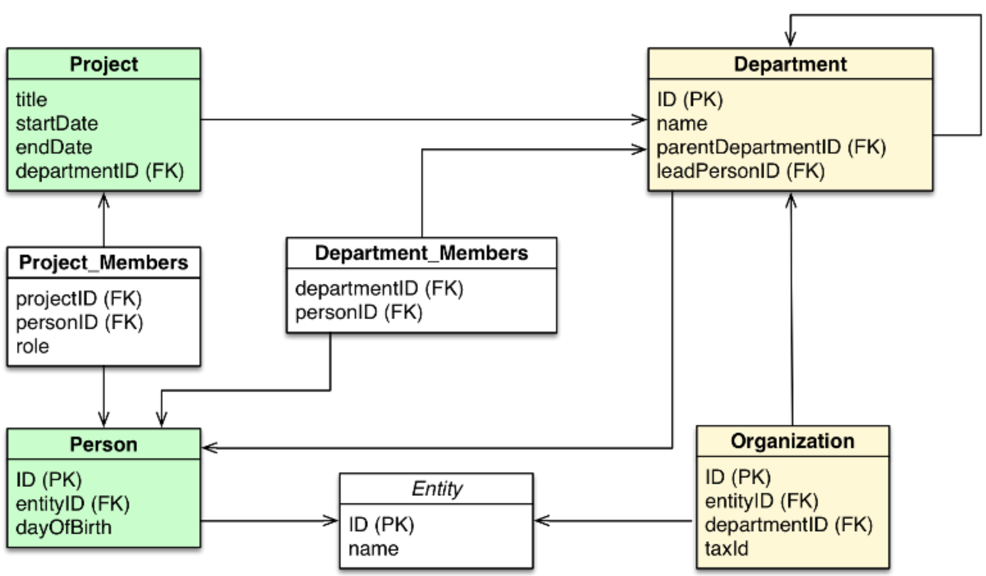


Relational schema
for movie data

Converting relational to graph-based

- Each row in a entity table becomes a **node**
- Each entity table becomes a **label** on nodes
- Columns on those tables become node **properties**
- Foreign keys become **relationships** to the corresponding nodes in the other table
- Join tables become relationships, columns on those tables become **relationship properties**

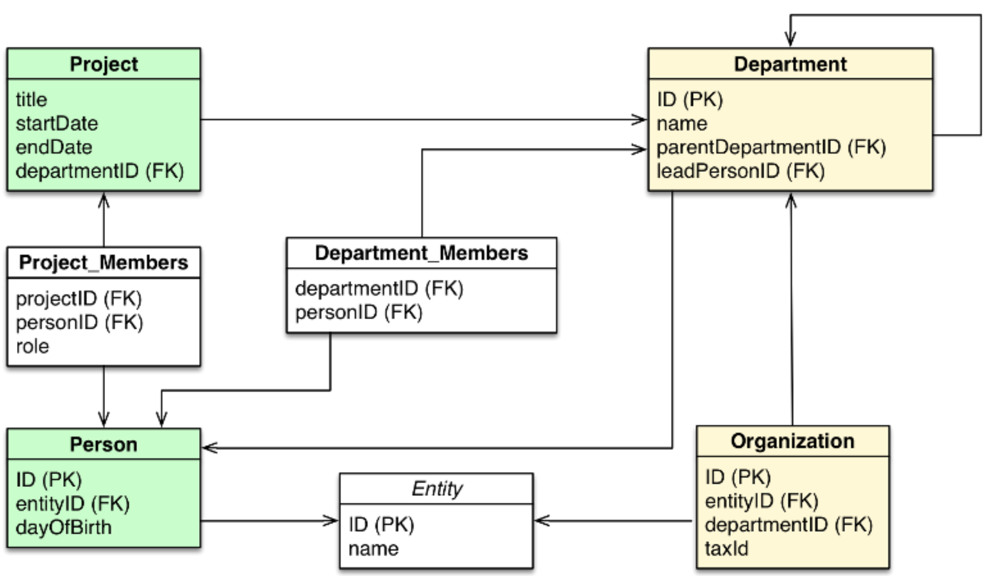
Relational vs graph schema



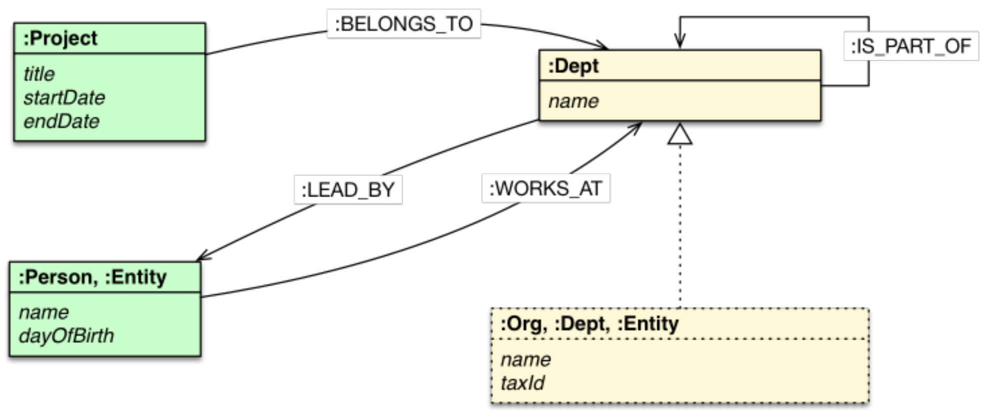
Relational

Graph-based

Relational vs graph schema



Relational



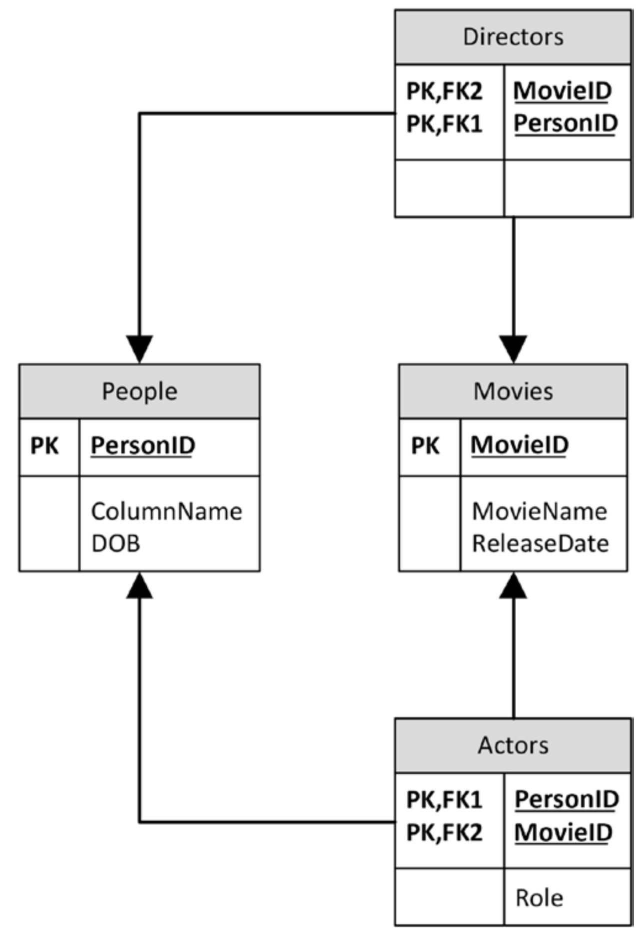
Graph-based

Graph databases
Issues with graphs in relational model

- SQL lacks the syntax to easily perform graph traversal
  - Especially traversals where the depth is unknown or unbounded
  - E.g., Kevin Bacon game, Erdos number
- Performance degrades quickly as we traverse the graph
  - Each level of traversal adds significantly to query response time.

Example graph

- SQL query to find actors who co-starred with Keanu Reeves

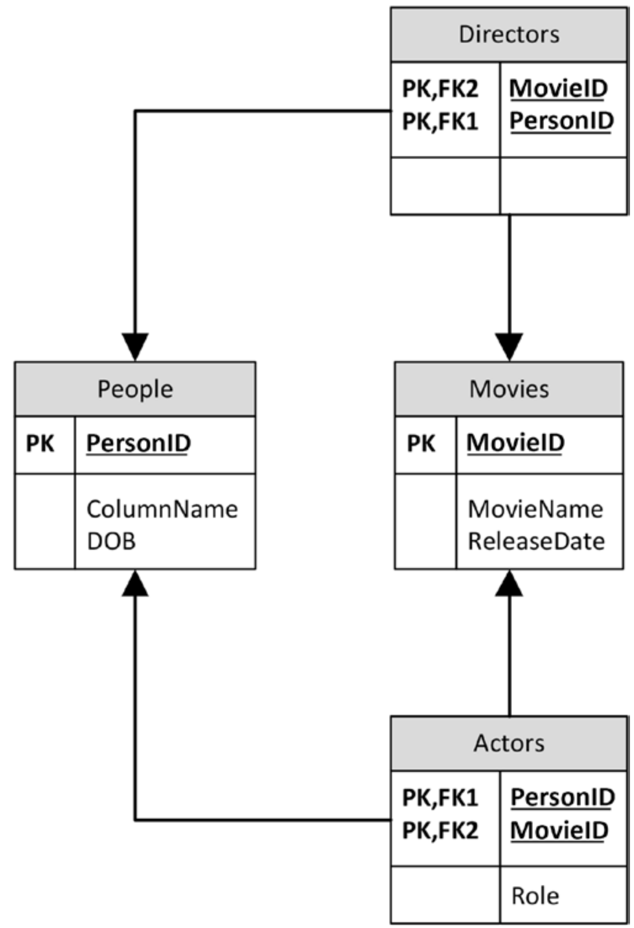


Relational schema for movie data

Example graph

- SQL query to find actors who co-starred with Keanu Reeves

```sql
SELECT p2.personname, m1.movieName
  FROM people p1
       JOIN actors a1 ON (p1.personid = a1.personid)
       JOIN movies m1 ON (a1.movieid = m1.movieid)
       JOIN actors a2 ON (a2.movieid = m1.movieid)
       JOIN people p2 ON (p2.personid = a2.personid)
 WHERE p1.personname = 'Keanu Reeves';
```



Relational schema for movie data

Issues with other database models for graph data

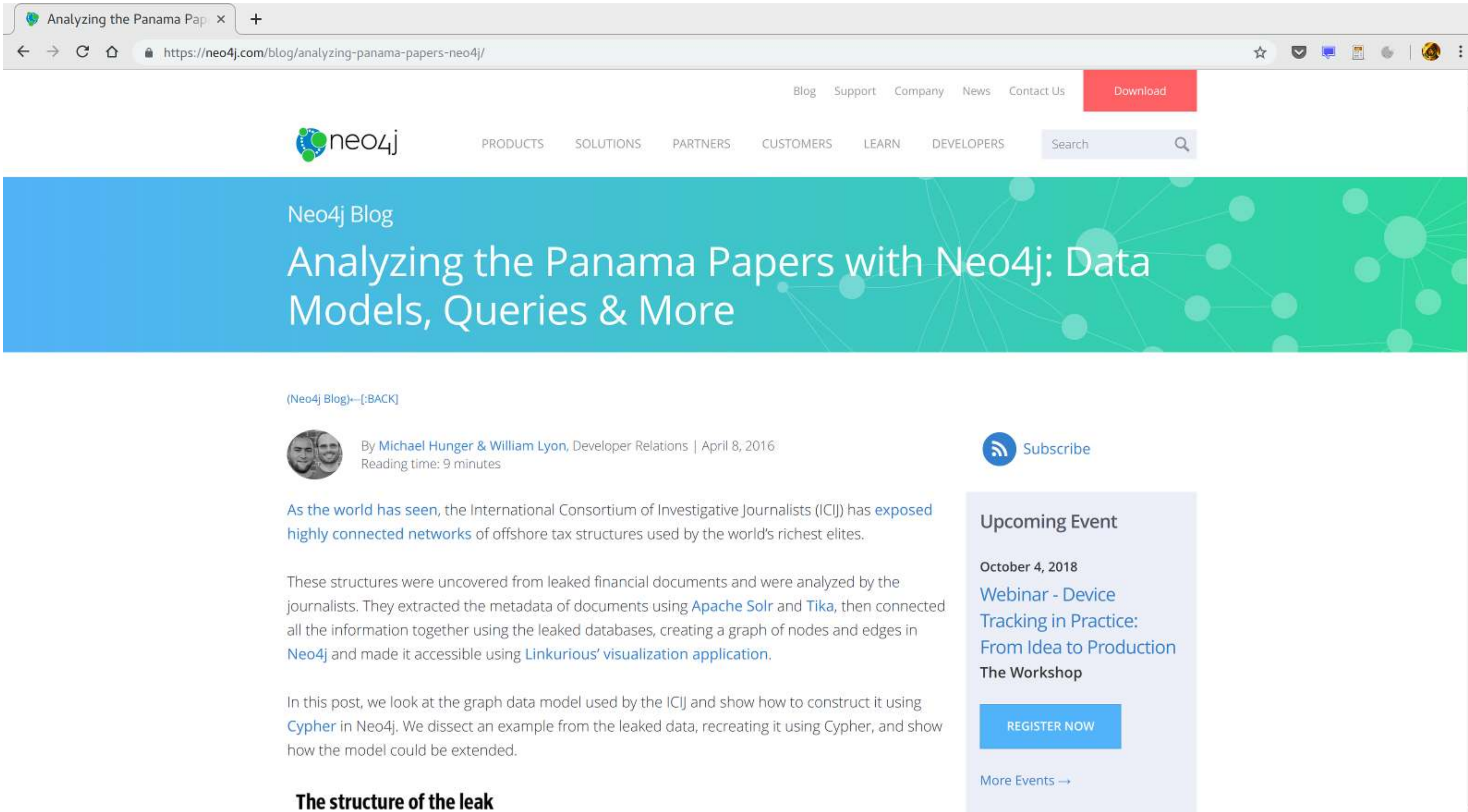
- Finding co-stars requires 5-way join
  - Add another 3 joins for each level deeper of query
- No syntax to allow arbitrary or unknown depth
- No syntax to expand the whole graph (unknown depth)
- Even with indexing, each join requires an index lookup for each actor and movie
- Fast, but memory-inefficient solution: Load the tables in their entirety into map structures in memory
- For key-value stores and document databases, graph must be traversed in application code

Neo4j syntax and examples

Example graph database: Neo4j

- Property graph model, nodes and edges both have properties
- Neo4j is the most popular graph database
- Written in Java
- Easily embedded in any Java application or run as a standalone server
- Supports billions of (graph) nodes, ACID compliant transactions, and multiversion consistency.
- Implements declarative graph query language Cypher
  - Query graphs in a way somewhat similar to SQL

Neo4j was used to quickly analyze the Panama Papers leak



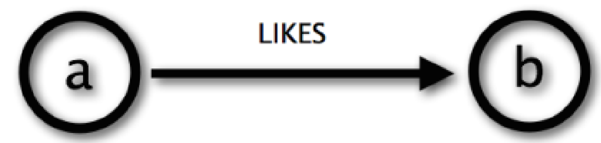
Neo4j uses Cypher graph query language

- Declarative query language (like SQL)
- ASCII-Art notation for nodes and edges
- Results are graphs as well
  - But can be displayed as tables

Cypher syntax: nodes/entities

- Nodes appear in parentheses: `(a)`, `(b)`
  - `a` and `b` are variables that can be referred to later in the query
  - Can access variables' properties, e.g. `a.name`
  - Can request nodes with a specific label using colon: `a:Person`
  - Other properties can be specified in braces: `(a:Person {name:"Mike"})`

Cypher using relationship 'likes'



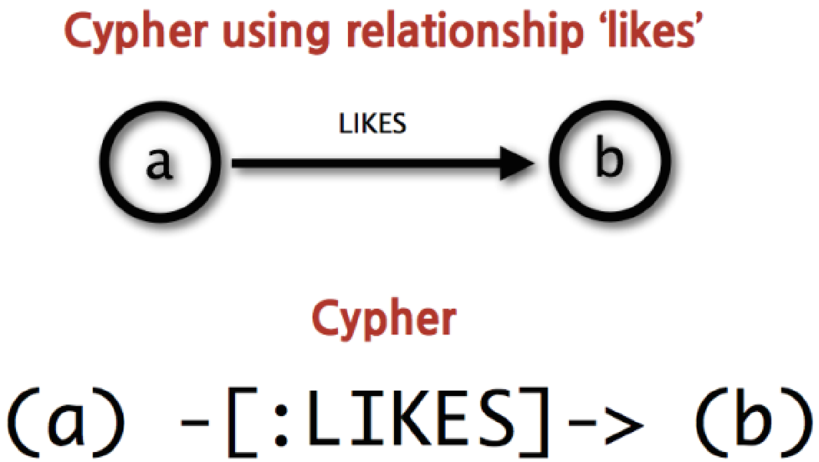
Cypher

`(a) -[:LIKES]-> (b)`

https://neo4j.com/developer/cypher-query-language/

Cypher syntax: edges/relationships

- Arrows for relationships `-->`
  - With variables and properties in square brackets: `-[r:LIKES]->`
  - Can include path length: `-[r:LIKES*..4]->`



https://neo4j.com/developer/cypher-query-language/

Cypher graph creation: `CREATE`

- Create a `Person` with `name` property of “You”:

https://neo4j.com/developer/cypher-query-language/

Cypher graph creation: `CREATE`

- Create a `Person` with `name` property of “You”:

```
CREATE (you:Person {name:"You"})
RETURN you
```

https://neo4j.com/developer/cypher-query-language/

Cypher graph querying: `MATCH`

- Query using `MATCH`
  - These are all equivalent for the one-node graph we just created

```
MATCH (n) RETURN n;
MATCH (n:Person) RETURN n;
MATCH (n:Person {name: "You"}) RETURN n;
```

https://neo4j.com/developer/cypher-query-language/

Cypher graph creation: MATCH and CREATE

- Add a new relationship between the existing node and a new node (you like neo, which is a Database with name property "Neo4j")

https://neo4j.com/developer/cypher-query-language/

Cypher graph creation: `MATCH` and `CREATE`

- Add a new relationship between the existing node and a new node (you like neo, which is a Database with name property "Neo4j")

```
MATCH  (you:Person {name:"You"})
CREATE (you)-[like:LIKE]->(neo:Database
{name:"Neo4j" })
RETURN you,like,neo
```

https://neo4j.com/developer/cypher-query-language/

Cypher graph creation: `FOREACH` to loop over things

- Add a new relationship between the existing node and several new nodes (create new friends Johan, Rajesh, Anna, Julia, and Andrew)

https://neo4j.com/developer/cypher-query-language/

Cypher graph creation: `FOREACH` to loop over things

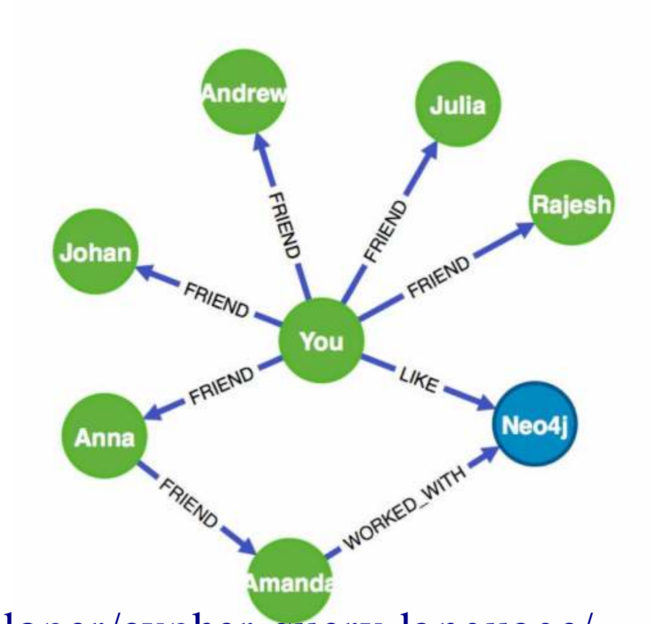
- Add a new relationship between the existing node and several new nodes (create new friends Johan, Rajesh, Anna, Julia, and Andrew)

```
MATCH (you:Person {name:"You"})
FOREACH (name in
["Johan","Rajesh","Anna","Julia","Andrew"] |
  CREATE (you)-[:FRIEND]->(:Person {name:name}))
```

https://neo4j.com/developer/cypher-query-language/

Cypher graph creation

- Create new relationship between two existing entities (Anna has worked_with Neo4j)

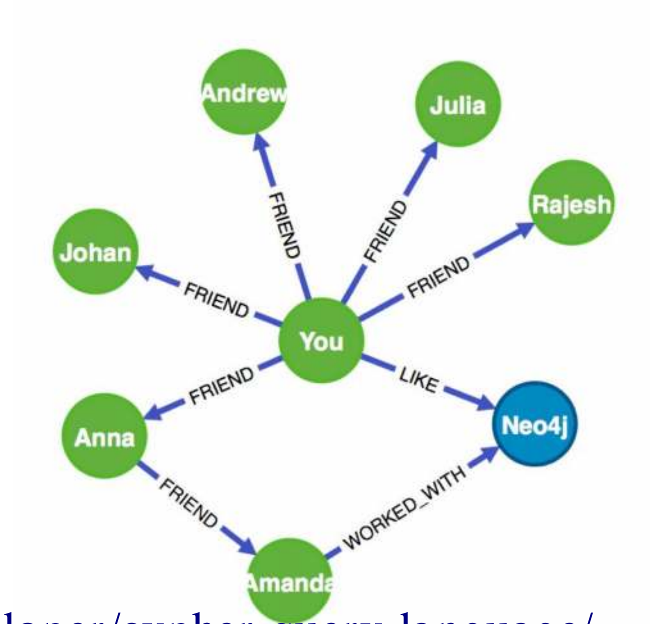


https://neo4j.com/developer/cypher-query-language/

Cypher graph creation

- Create new relationship between two existing entities (Anna has worked_with Neo4j)

```
MATCH (neo:Database {name:"Neo4j"})
MATCH (anna:Person {name:"Anna"})
CREATE (anna)-[:FRIEND]->(:Person:Expert
{name:"Amanda"})-[:WORKED_WITH]->(neo)
```



https://neo4j.com/developer/cypher-query-language/

Cypher graph querying

- Find shortest friend-of-a-friend path to someone in your network who can help you learn Neo4j

https://neo4j.com/developer/cypher-query-language/

Cypher graph querying

- Find shortest friend-of-a-friend path to someone in your network who can help you learn Neo4j

```
MATCH (you {name:"You"})
MATCH (expert)-[:WORKED_WITH]->(db:Database
{name:"Neo4j"})
MATCH path = shortestPath( (you)-[:FRIEND*..5]-
(expert) )
RETURN db,expert,path
```

https://neo4j.com/developer/cypher-query-language/

Example graph database: Neo4j
Create example graph

```
CREATE (TheMatrix:Movie {title:'The Matrix',
released:1999, tagline:'Welcome to the Real World'})

CREATE (JohnWick:Movie {title:'John Wick', released:2014,
tagline:'Silliest Keanu movie ever'})

CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})

CREATE (AndyW:Person {name:'Andy Wachowski', born:1967})

CREATE

 (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),

 (Keanu)-[:ACTED_IN {roles:['John Wick']}]->(JohnWick),

 (AndyW)-[:DIRECTED]->(TheMatrix)
```

Example graph database: Neo4j
Retrieve info on one node

```
MATCH (keanu:Person {name:"Keanu Reeves"})
RETURN keanu;

+-----------------------------------------+
| keanu                                   |
+-----------------------------------------+
| Node[1]{name:"Keanu Reeves",born:1964} |
+-----------------------------------------+
```

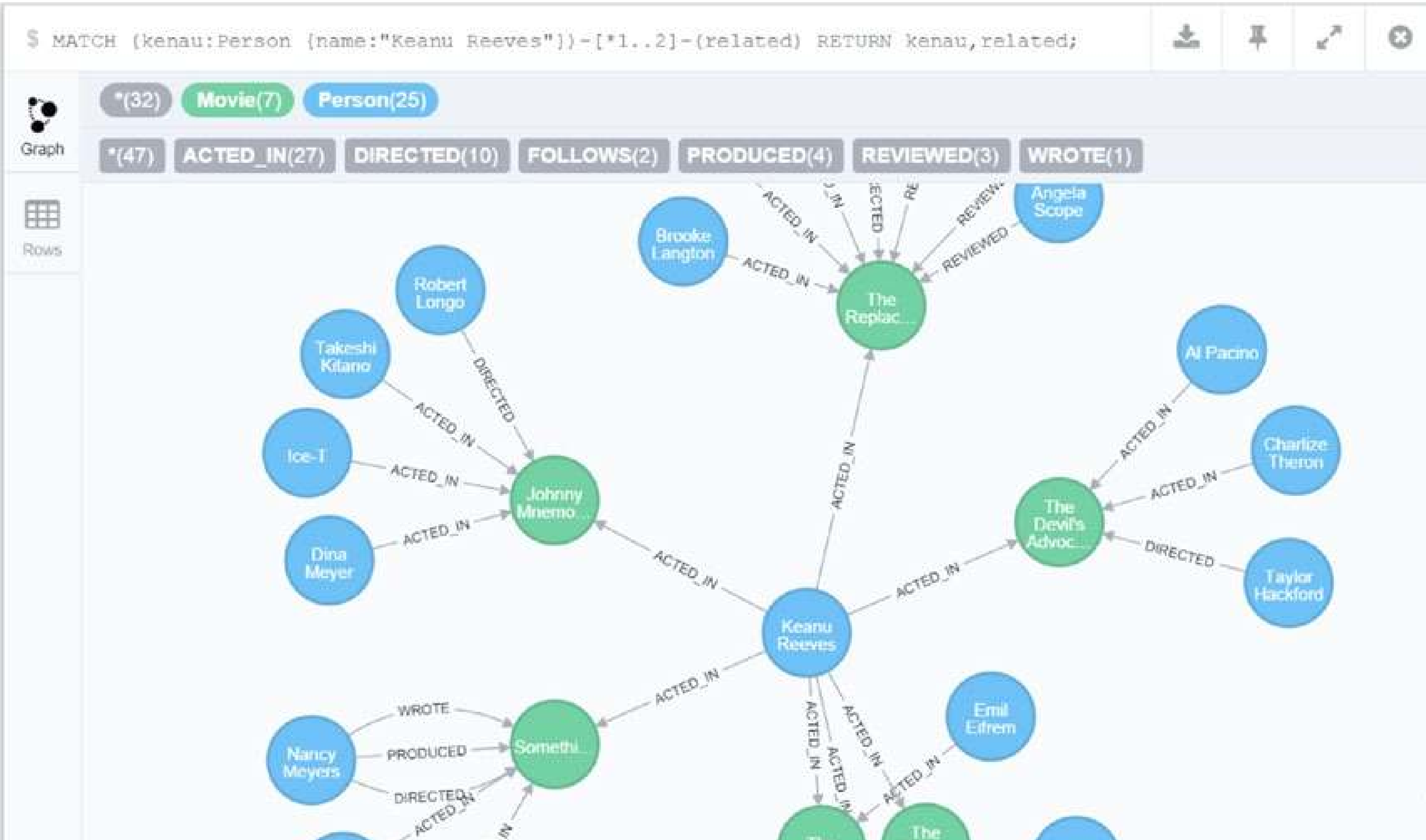
Bigger graph database in Neo4j
Find all co-stars of Keanu

```
MATCH (kenau:Person {name:"Keanu Reeves"})-
[:ACTED_IN]→(movie)<-[:ACTED_IN]-(coStar) RETURN
coStar.name;

+-----------------------+
| coStar.name           |
+-----------------------+
| "Jack Nicholson"      |
| "Diane Keaton"        |
| "Dina Meyer"          |
| "Ice-T"               |
| "Takeshi Kitano"      |
```

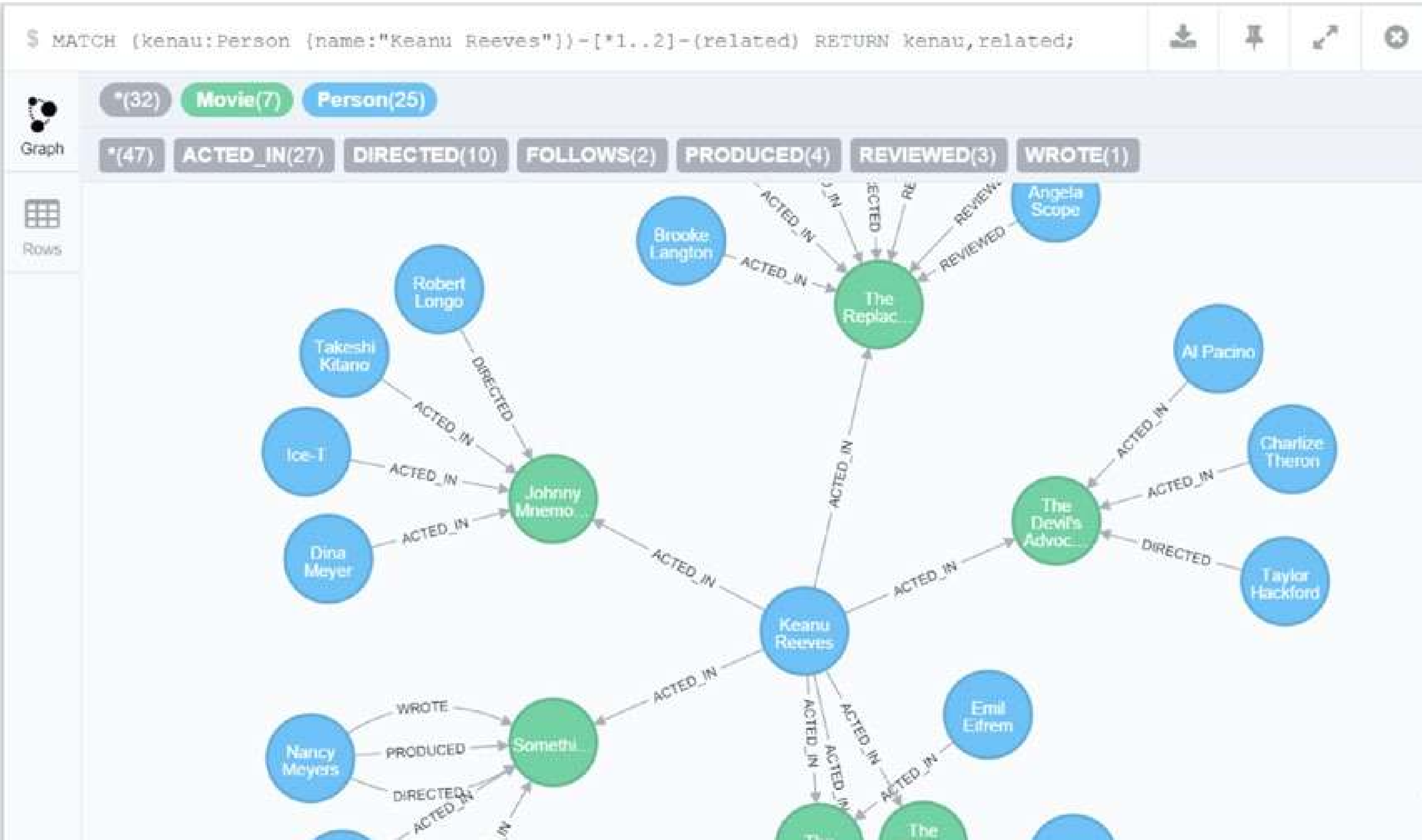
Bigger graph database in Neo4j
Find all nodes within 2 hops of Keanu

```
MATCH (kenau:Person {name:"Keanu Reeves"})-[*1..2]-
(related) RETURN distinct related;

+-------------------------------------------------------|
| related                                               |
+-------------------------------------------------------|
| Node[0]{title:"The Matrix",released:1999, tagline:…}  |
| Node[7]{name:"Joel Silver",born:1952}                 |
| Node[5]{name:"Andy Wachowski",born:1967}              |
| Node[6]{name:"Lana Wachowski",born:1965}              |
| …                                                     |
```

Bigger graph database in Neo4j
Results are graphs too



Bigger graph database in Neo4j Demo



Graph database internals: Index-free adjacency

- Graph processing can be performed on databases irrespective of their internal storage format
  - It is a logical model, not a specific implementation
- Efficient real-time graph processing requires moving through graph without index lookups
  - Referred to as index-free adjacency
- In RDBMS, indexes allow logical key values to be translated to a physical addresses
  - Typically, three or four logical IO operations are required to traverse a B-Tree index
  - Plus another lookup to retrieve the value
  - Can be cached, but usually some disk IO required
- In a native graph database utilizing index-free adjacency, each node knows the physical location of all adjacent nodes
  - So no need to use indexes to efficiently navigate the graph

Graph compute engines: Add graph interface on other models

- Implements efficient graph processing algorithms
- Exposes graph APIs
- Doesn't necessarily store data in index free adjacency graph
  - Usually designed for batch processing of most or all of a graph
- Significant examples:
  - Apache Giraph: graph processing on Hadoop using MapReduce
  - GraphX: graph processing part in Spark (part of Berkeley Data Analytic Stack)
  - Titan: graph processing on Big Data storage engines like Hbase and Cassandra

Graph databases
Strengths and weaknesses

- Strengths
  - Joins are precomputed
  - Flexible schema
  - Fast and scalable
- Weaknesses
  - Harder to execute queries not embodied in relationships

Document databases

Document databases

- Non-relational database that stores data as structured documents
  - Usually XML or JSON formats
- Doesn’t imply anything specific beyond the document storage model
- Could implement ACID transactions, etc
  - Most provide modest transactional support
- Try to remove object-relational impedance mismatch
- Easy to incorporate media in documents

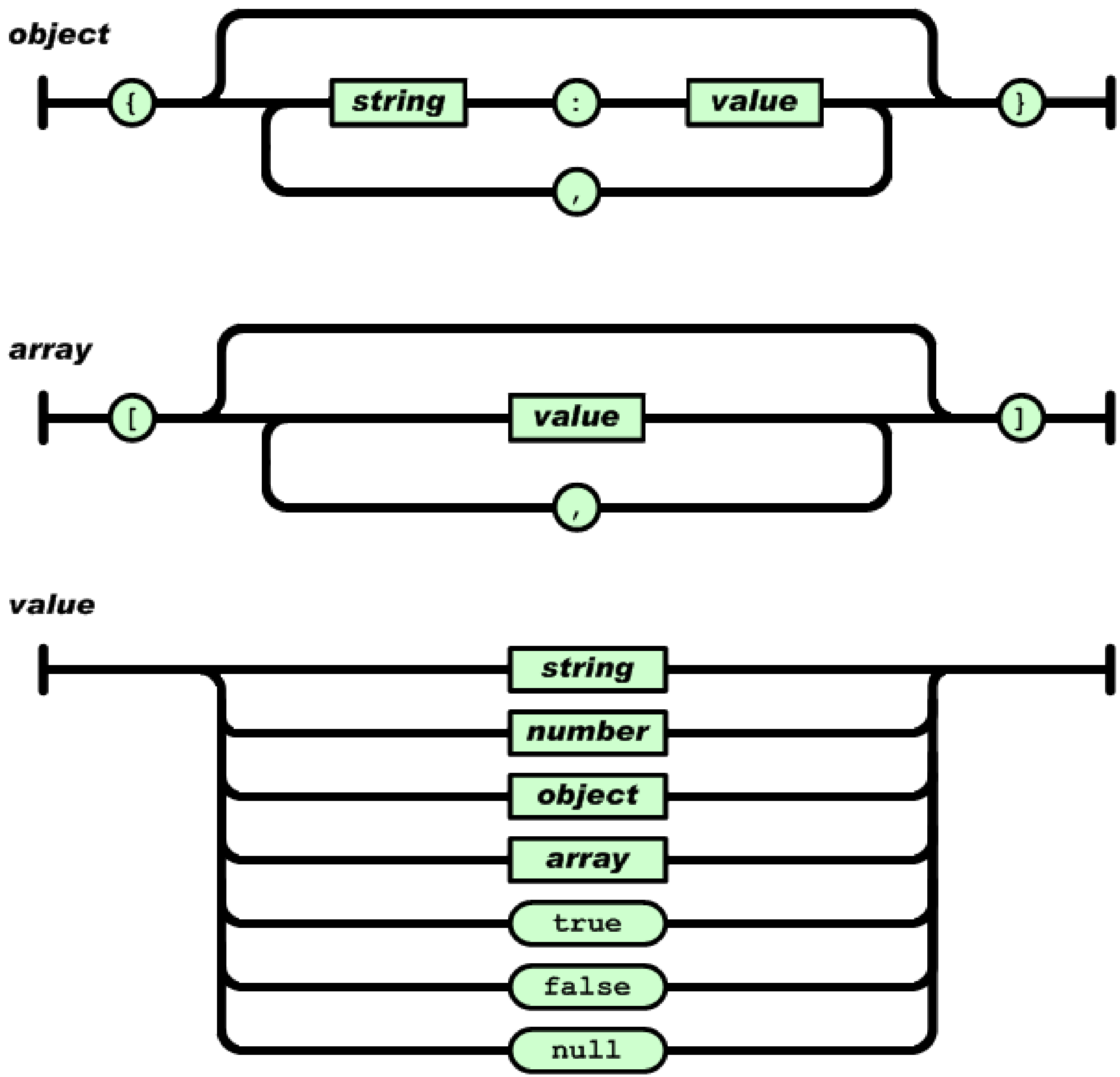
JSON-based document databases flourished starting in 2008

- Address the conflict between object-oriented programming and the relational database model
- Self-describing document formats could be interrogated independently of the program that had created them
- Aligned well with the dominant web-based programming paradigms

JSON databases

- Not a single specification, just store data in JSON format, but usually
- Basic unit of storage is the Document ≈ a row in an RDBMS
  - One or more key-value pairs
  - May also contain nested documents and arrays
  - Arrays may also contain documents, creating complex hierarchical structure
- A collection or bucket is a set of documents sharing some common purpose
  - ≈ a relational table
  - Documents in a collection don't have to be the same type
- Could implement 3rd normal form schema
  - But usually model data in a smaller number of collections
  - With nested documents representing master-detail relationships

JSON format


object
{
string
:
value
,
}
array
[
value
,
]
value
string
number
object
array
true
false
null

Example JSON structure (from homework)


```
{
  "url": "https:\/\/farm5.staticflickr.com\/4469\/23531804118_fce6162cd1.jpg",
  "response": {
    "labelAnnotations": [
      {
        "score": 0.85065764188766,
        "mid": "\/m\/07yv9",
        "description": "vehicle"
      },
      ...
    ],
    "webDetection": {
      "fullMatchingImages": [
        {
          "url": "https:\/\/farm6.staticflickr.com\/5079\/7403056606_a09f6f670e_b.jpg"
        },
        ...
      ],
      "pagesWithMatchingImages": [
        {
          "url": "http:\/\/picssr.com\/photos\/32622429@N02\/favorites\/page109?nsid=32622429@N02"
        },
        ...
      ],
      "webEntities": [
        {
          "score": 3.0824000835419,
          "entityId": "\/m\/02vqfm",
          "description": "Coffee"
        },
        ...
      ],
      "partialMatchingImages": [
        {
          "url": "https:\/\/farm6.staticflickr.com\/5079\/7403056606_a09f6f670e_b.jpg"
        },
        ...
      ]
    },
```

MongoDB syntax and examples

MongoDB Create operations

```
db.users.insertOne(
   {
      name: "sue",
      age: 26,
      status: "pending"
   }
)
```

collection

field: value

field: value

field: value

document

- Several commands: insertOne(), insertMany(), and save()
- save() will update an existing document if found or insert a new one if not

https://docs.mongodb.com/manual/crud/#crud

MongoDB Search operations

```
db.users.find(              ← collection
   { age: { $gt: 18 } },    ← query criteria
   { name: 1, address: 1 }  ← projection
).limit(5)                  ← cursor modifier
```

- Select documents matching certain criteria
- Project to only the fields of interest
- Cursor modifiers include count(), limit(), skip(), sort()

https://docs.mongodb.com/manual/crud/#crud

MongoDB Aggregation

- Aggregation allows you to perform more complex queries
- Three interfaces
  - Aggregation Pipeline
  - Map-Reduce
  - Single-purpose aggregation operations

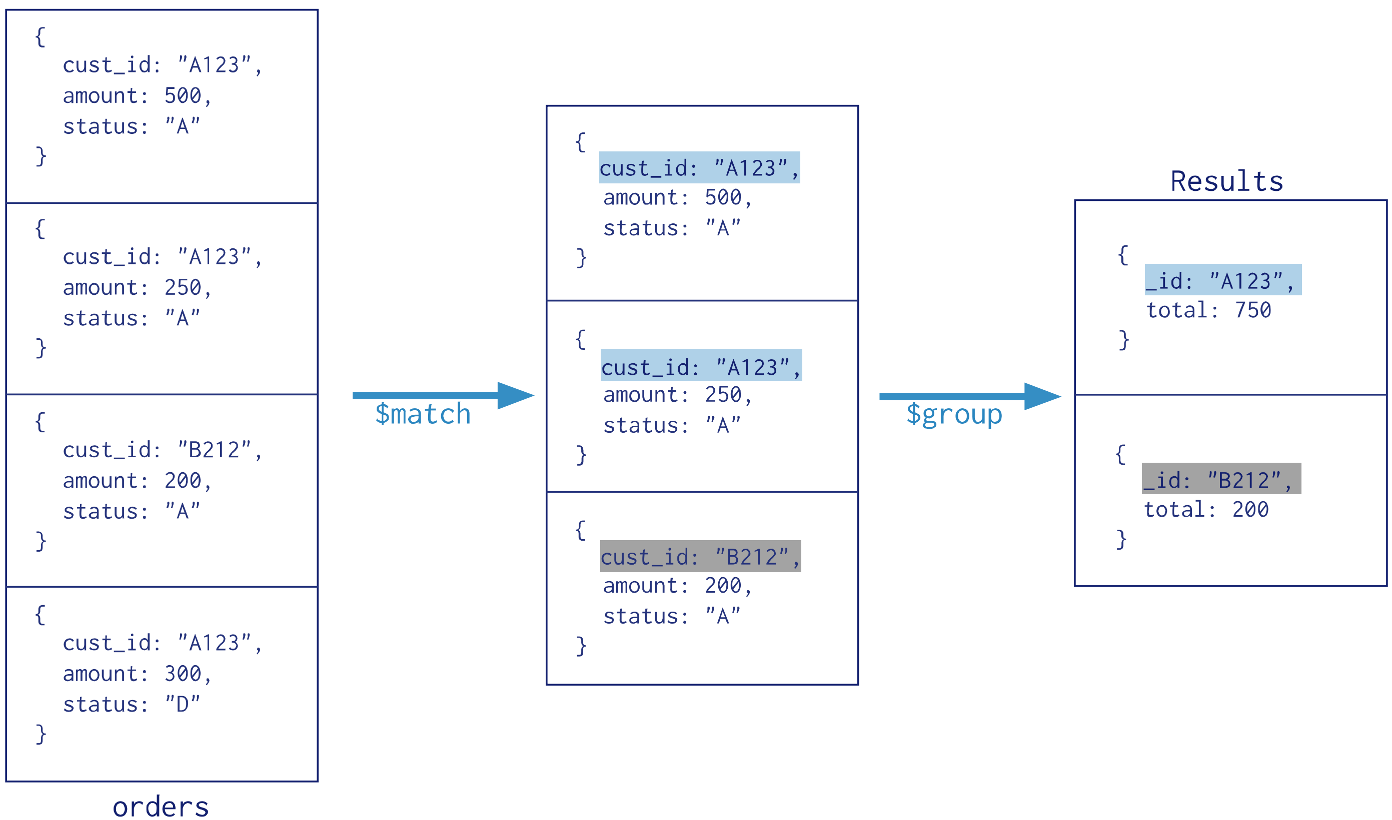
MongoDB Aggregation pipeline ex

Collection

$match stage

$group stage

```
db.orders.aggregate( [
    { $match: { status: "A" } },
    { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```



https://docs.mongodb.com/manual/aggregation/

MongoDB Aggregation pipeline

- Provide a sequence of **stages** of processing
  - Each stage uses one command below, modifies or filters the selected documents, results fed into next
  - Like a pipeline in a unix shell

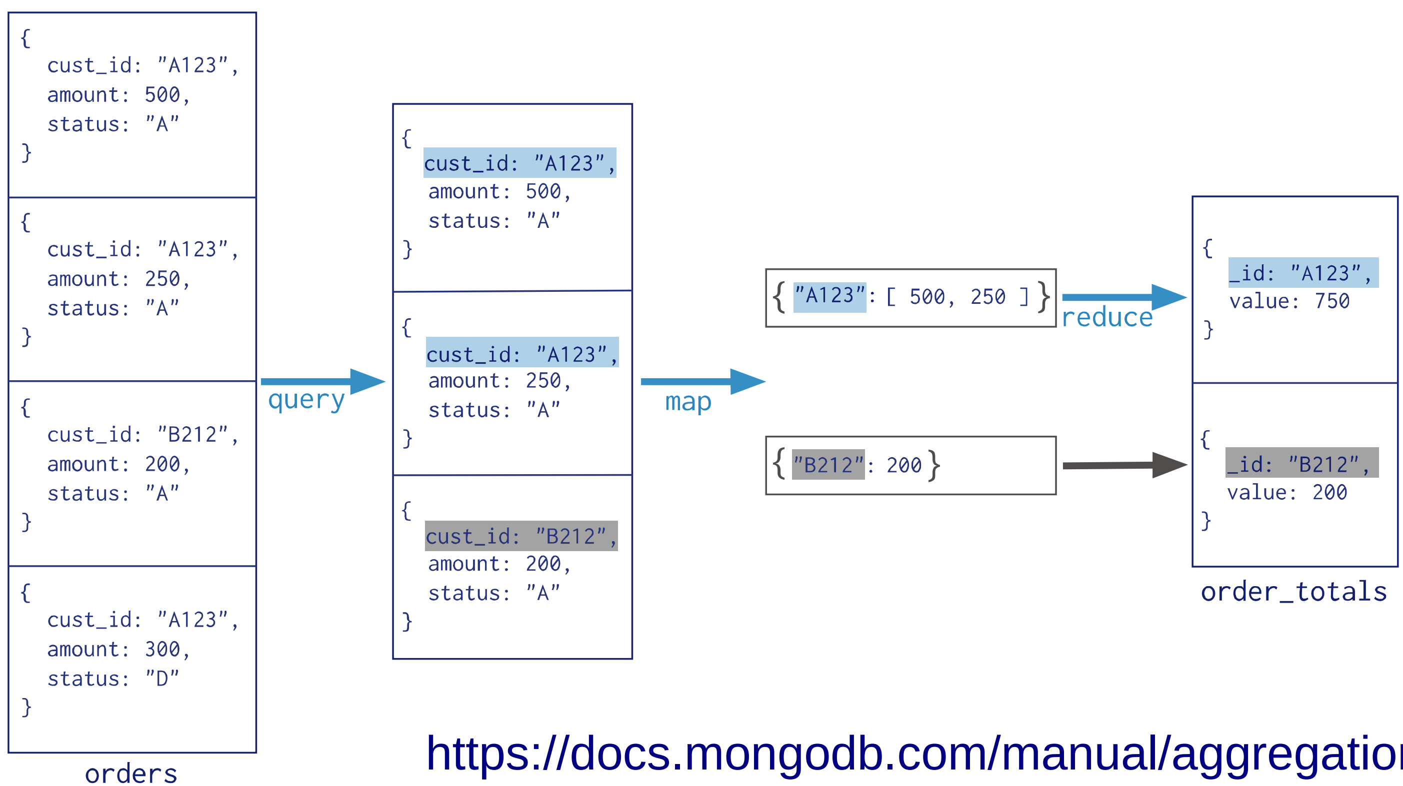
Operator	Operation
$project	Select a subset of fields from documents
$match	Discard documents not matching provided criteria
$group	Aggregate documents based on keys with various summaries
$sort	Order the results
$skip	Omit the documents from the beginning of the results
$limit	Limit results to only this number
$unwind	Enter arrays and process nested documents

https://www.tutorialspoint.com/mongodb/mongodb_aggregation.htm

MongoDB Map-Reduce ex

Collection

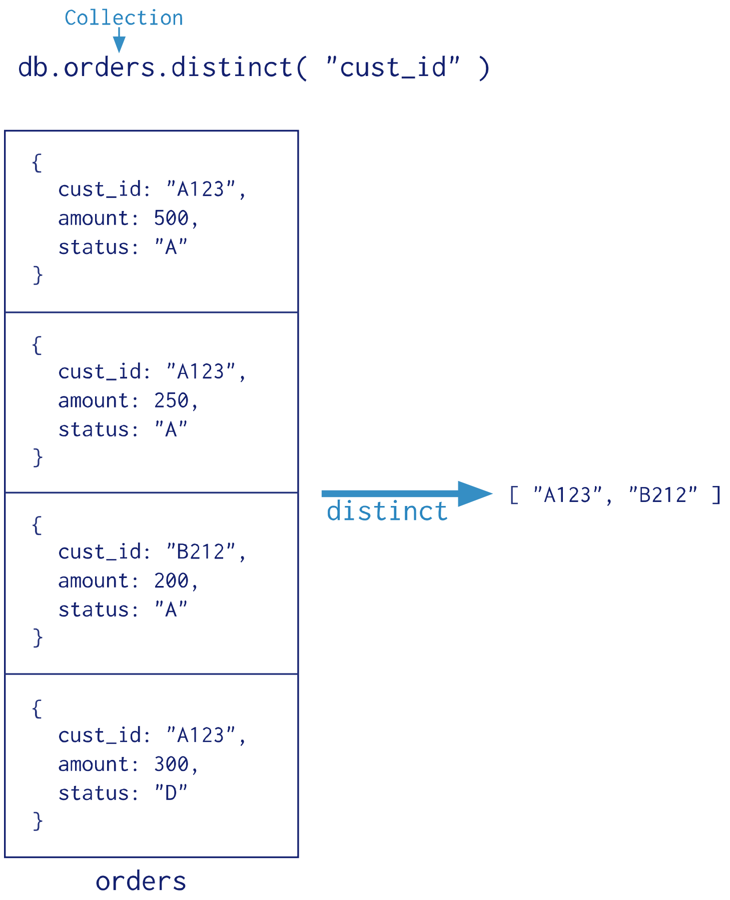
```
db.orders.mapReduce(
   map    →  function() { emit( this.cust_id, this.amount ); },
   reduce →  function(key, values) { return Array.sum( values ) },
             {
   query  →    query: { status: "A" },
   output →    out: "order_totals"
             }
           )
```



https://docs.mongodb.com/manual/aggregation/#map-reduce

MongoDB Single-purpose aggregation functions

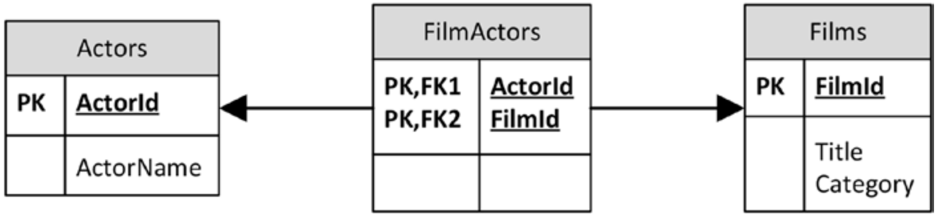
- Less flexible
- But easier to use
- Two functions:
  - db.collection.count()
  - db.collection.distinct()



https://docs.mongodb.com/manual/aggregation/#single-purpose-aggregation-operations

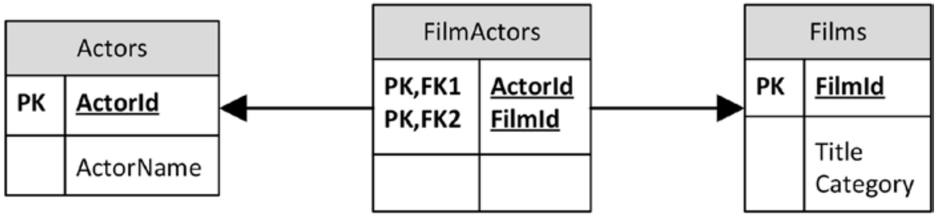
Example JSON database

- Relational schema



Example JSON database

- Relational schema



- Example JSON database
- Uses typical "**Document embedding**"
- Mirrors OO design

```
{ "_id" : 97, "Title" : "BRIDE INTRIGUE",
            "Category" : "Action",
    "Actors" :
      [ { "actorId" : 65, "Name" : "ANGELA HUDSON" } ]
}

{ "_id": 115,"Title": "CAMPUS REMEMBER",
            "Category": "Action",
    "Actors" :
       [
        { "actorId": 8,"Name": "MATTHEW JOHANSSON" },
        { "actorId": 45,"Name": "REESE KILMER" },
        { "actorId": 168,"Name": "WILL WILSON" }
       ]
}

{ "_id" : 105, "Title" : "BULL SHAWSHANK",
            "Category" : "Action",
     "Actors" :
       [ { "actorId" : 2, "Name" : "NICK WAHLBERG" },
         { "actorId" : 23, "Name" : "SANDRA KILMER" } ]
}
```

Film Document

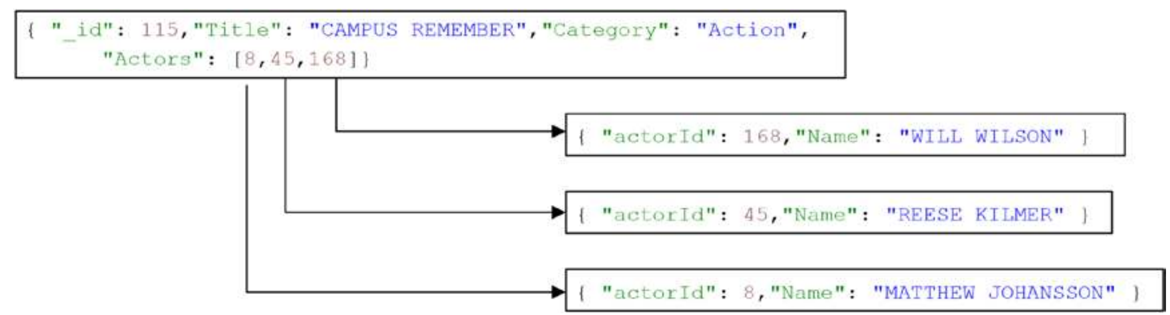
Actor document

Array of actors

Collection

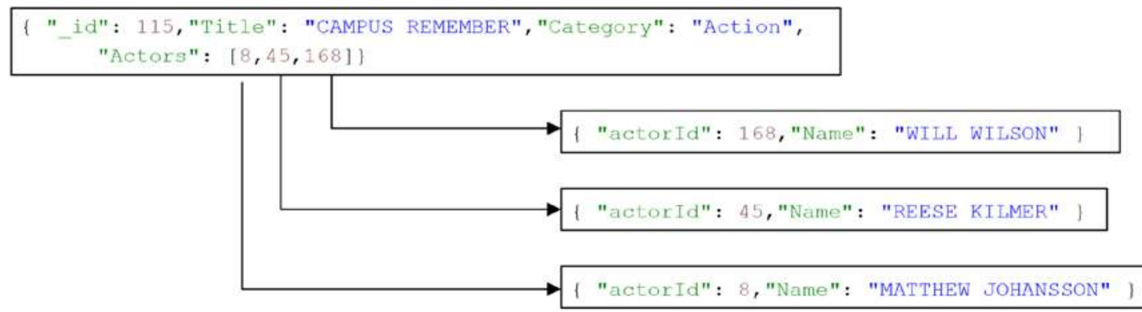
Example JSON database Document linking

- Could instead use **document linking** to list of actor documents

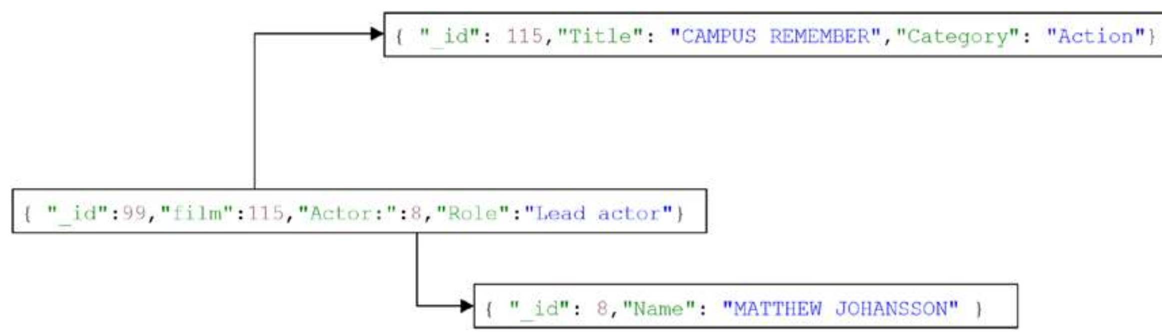


Example JSON database Document linking

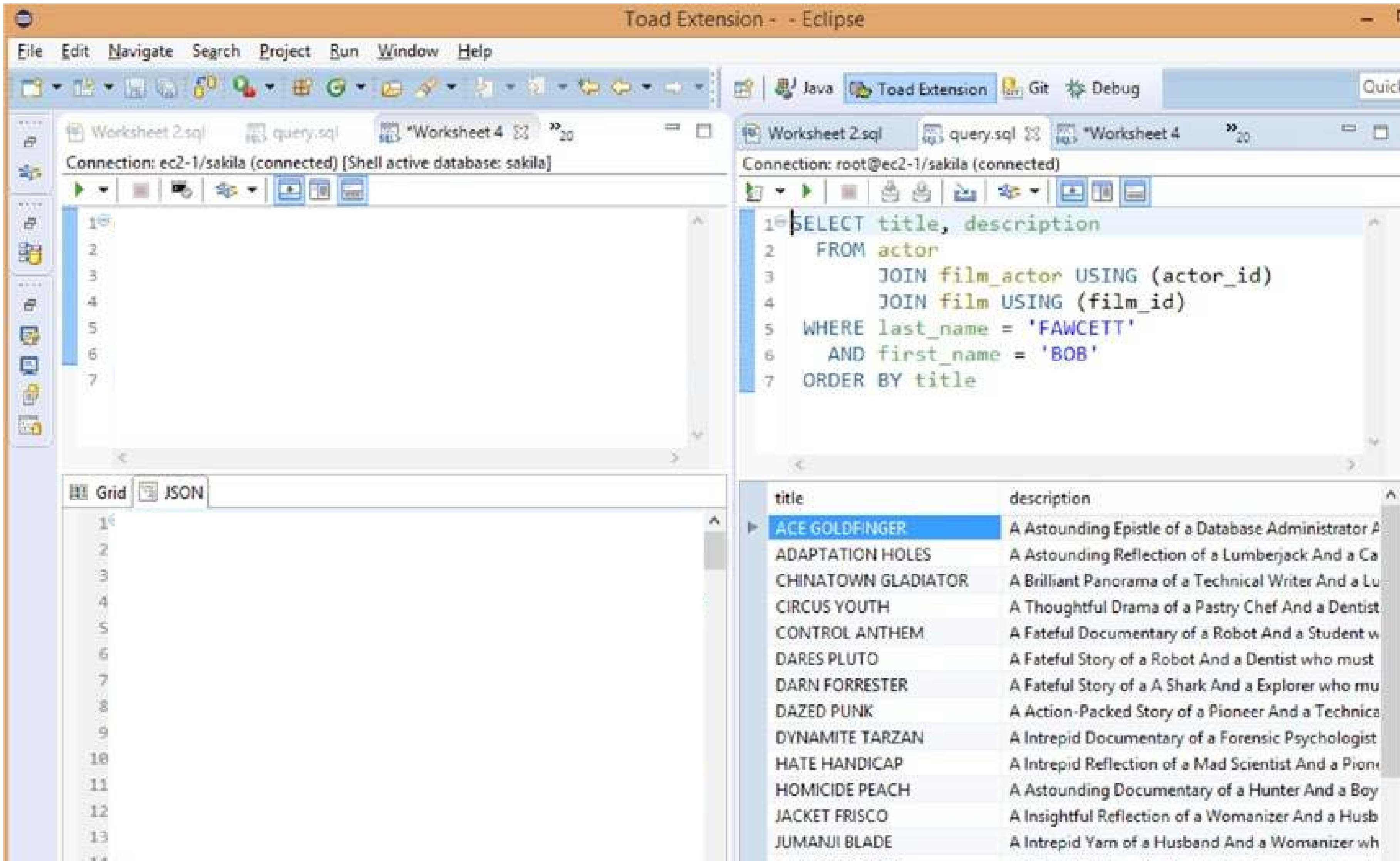
- Could instead use **document linking** to list of actor documents



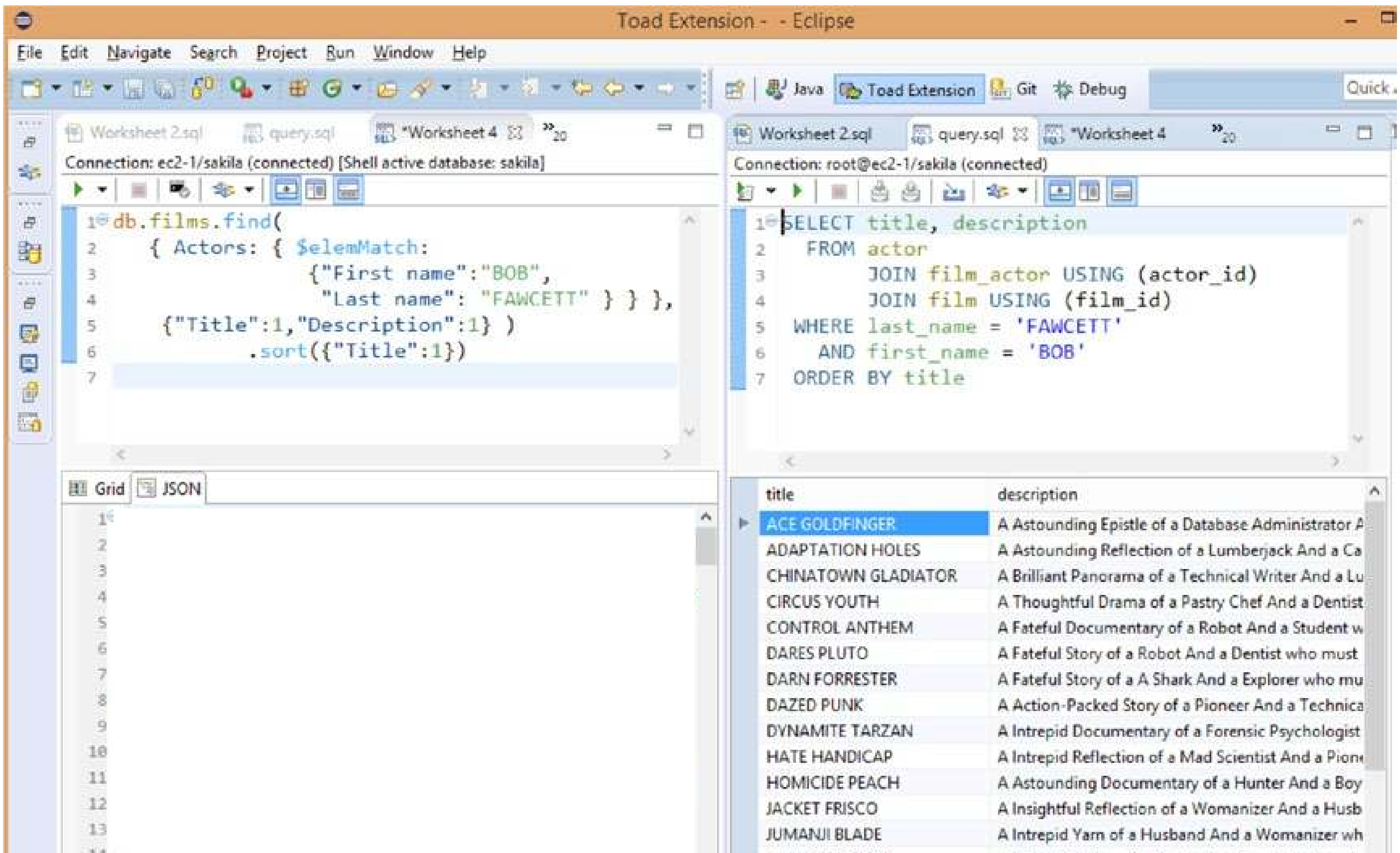
- Or could use **relational-style document linking**, closer to 3rd normal form
- Less natural for document DB because of lack of joins



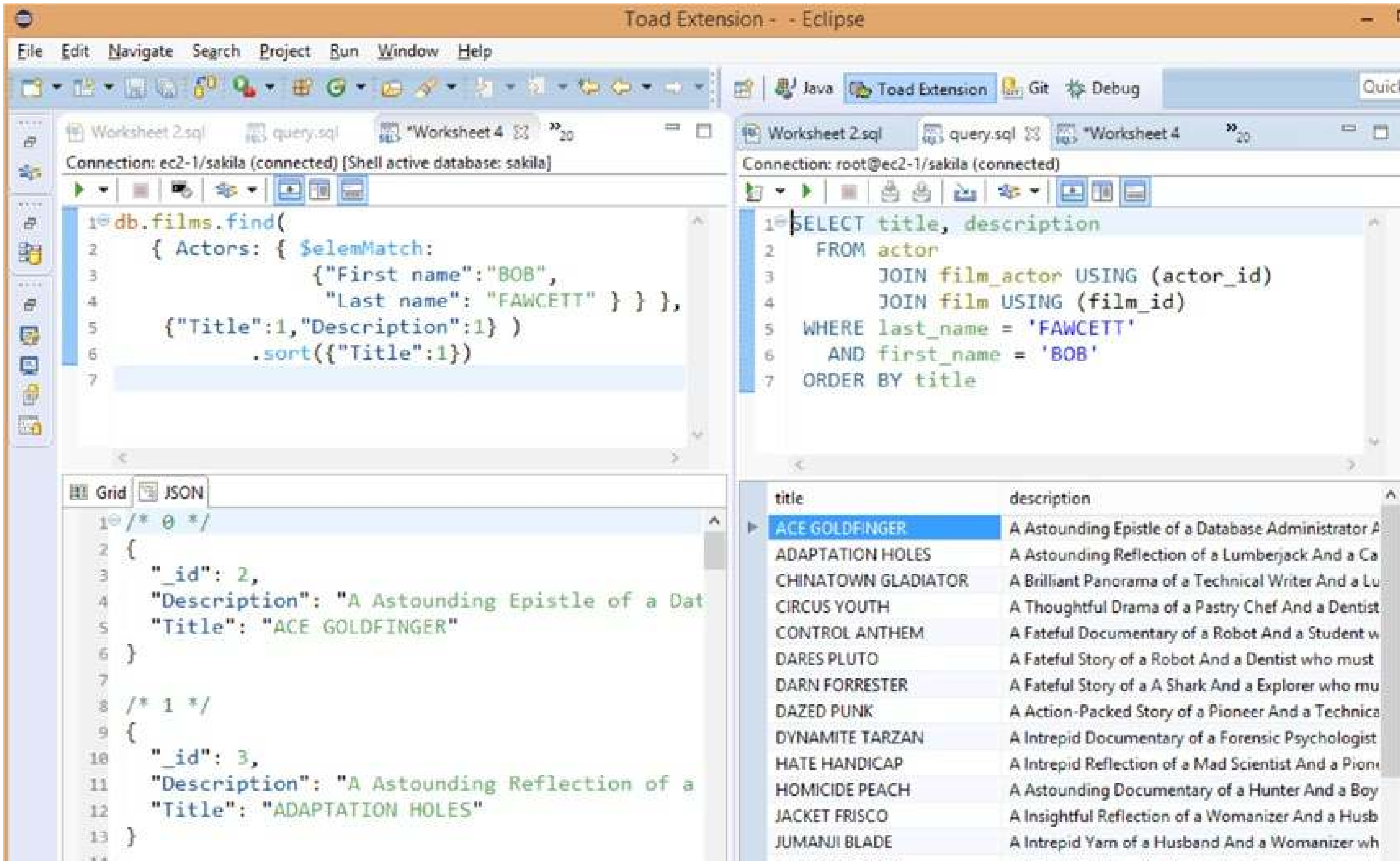
Example document DB: MongoDB Query with JavaScript



Example document DB: MongoDB Query with JavaScript



Example document DB: MongoDB Query with JavaScript



Document database summary

- Like a key-value store, but with self-documenting values
- Like a traditional RDBMS, but more scalable, less mis-match with object-oriented programming
- Many types of databases are adding support for JSON, so a "JSON document database" might soon be a feature of other databases instead of a distinct type of database

Document databases Strengths and weaknesses

- Strengths
  - Self-documenting schema
  - Easier for non-programmers to query
  - Can offer availability instead of strict consistency
- Weaknesses
  - Typically weak transactional support
  - Joins implemented in application code

Column databases

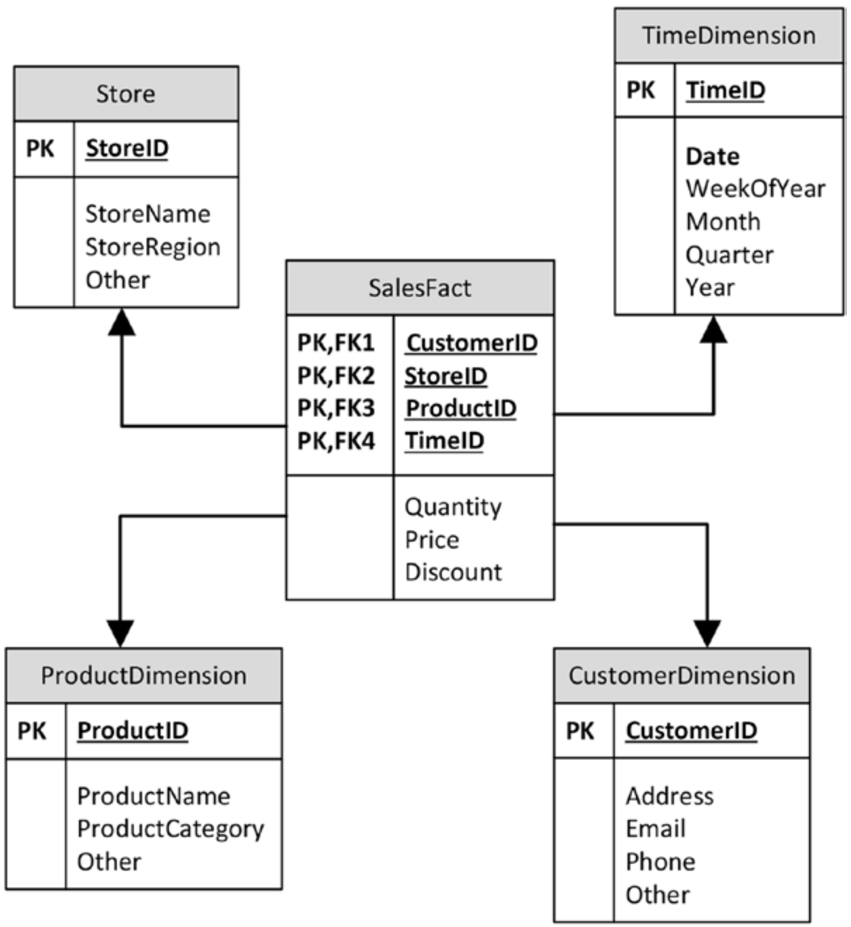
Column databases

- RDBMs is tuned for **OLTP**: OnLine Transaction Processing
  - Data in relations are organized by row (tuple)
  - All data in a row are stored together
- Data warehouses used for analytics present a different workload: **OLAP**: OnLine Analytic Processing
  - Aggregate information over many records to provide insight into trends
  - Organizing relations by column has several advantages

Data warehousing

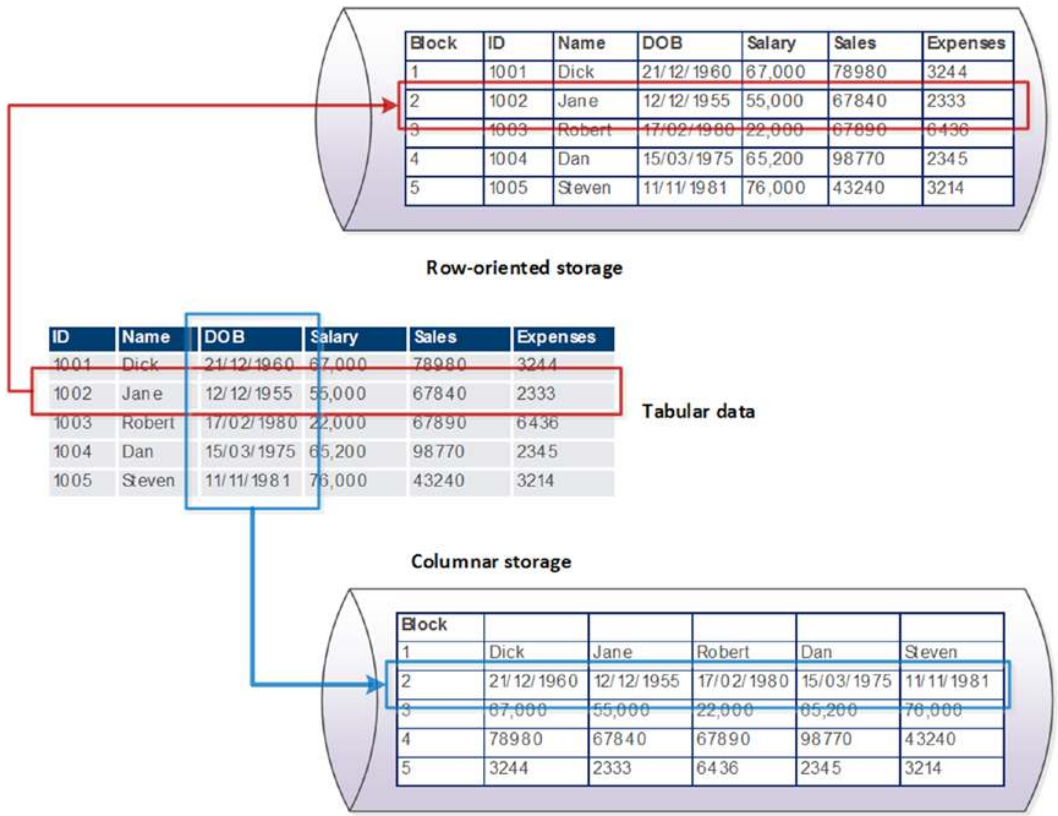
- In the 1970s, OLTP happened in real time, reports were generated in batches overnight
- In the 1980s and 90s, RDBMs started to be used for generating reports in real-time in parallel to the OLTP systems
  - Known as data warehouses
  - **Star schema** de-normalizes data somewhat to provide real-time responses

Star Schema
For RDBMS data warehouse

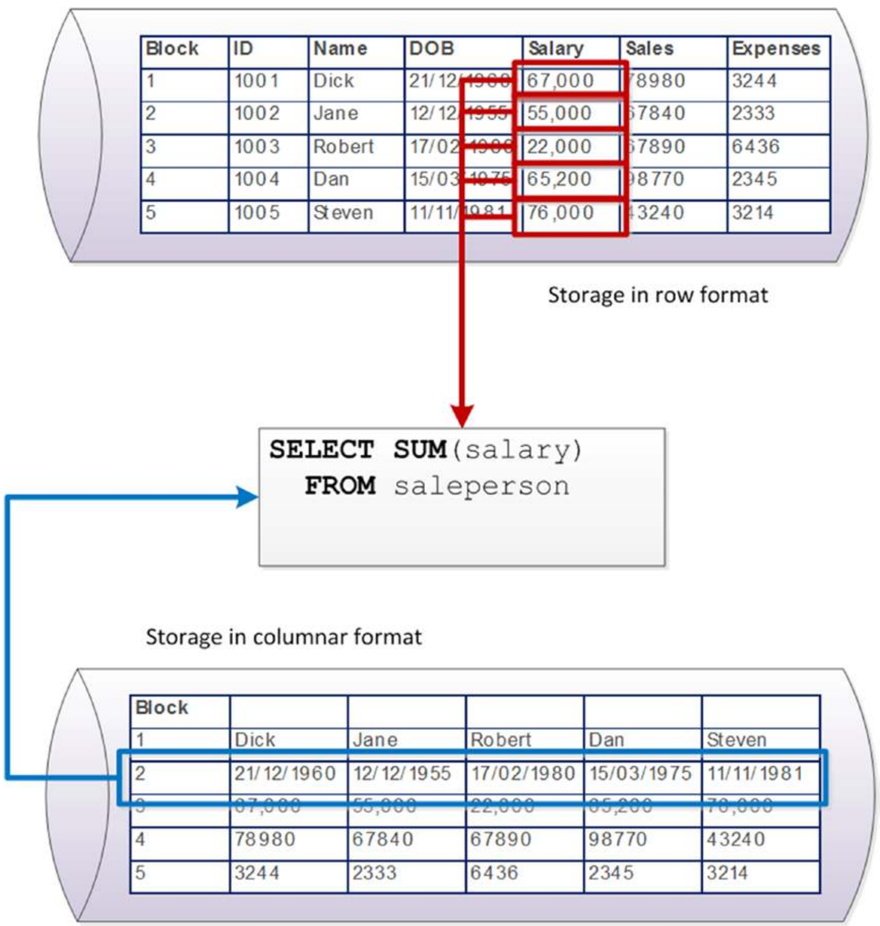


- Large central "fact" table
  - Measurements or metrics of each event
- Smaller "dimension" tables
  - Attributes to describe facts
- Still CPU and I/O intensive to use

Instead: Column Databases
Store relations by column

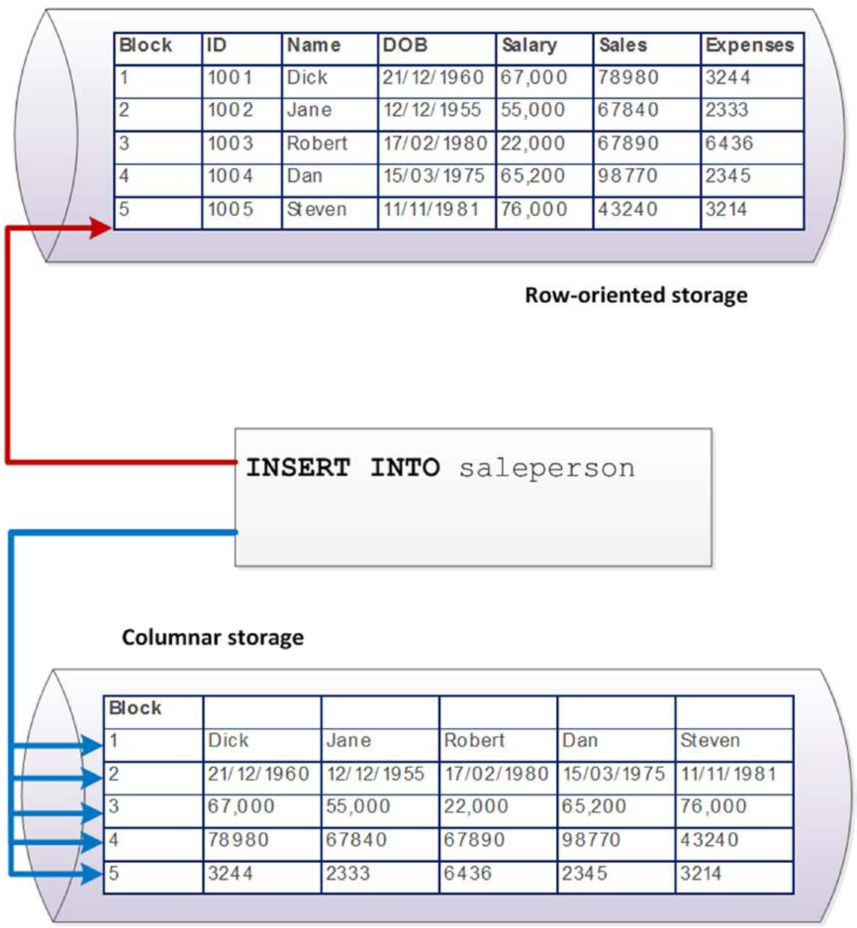


Column database advantages



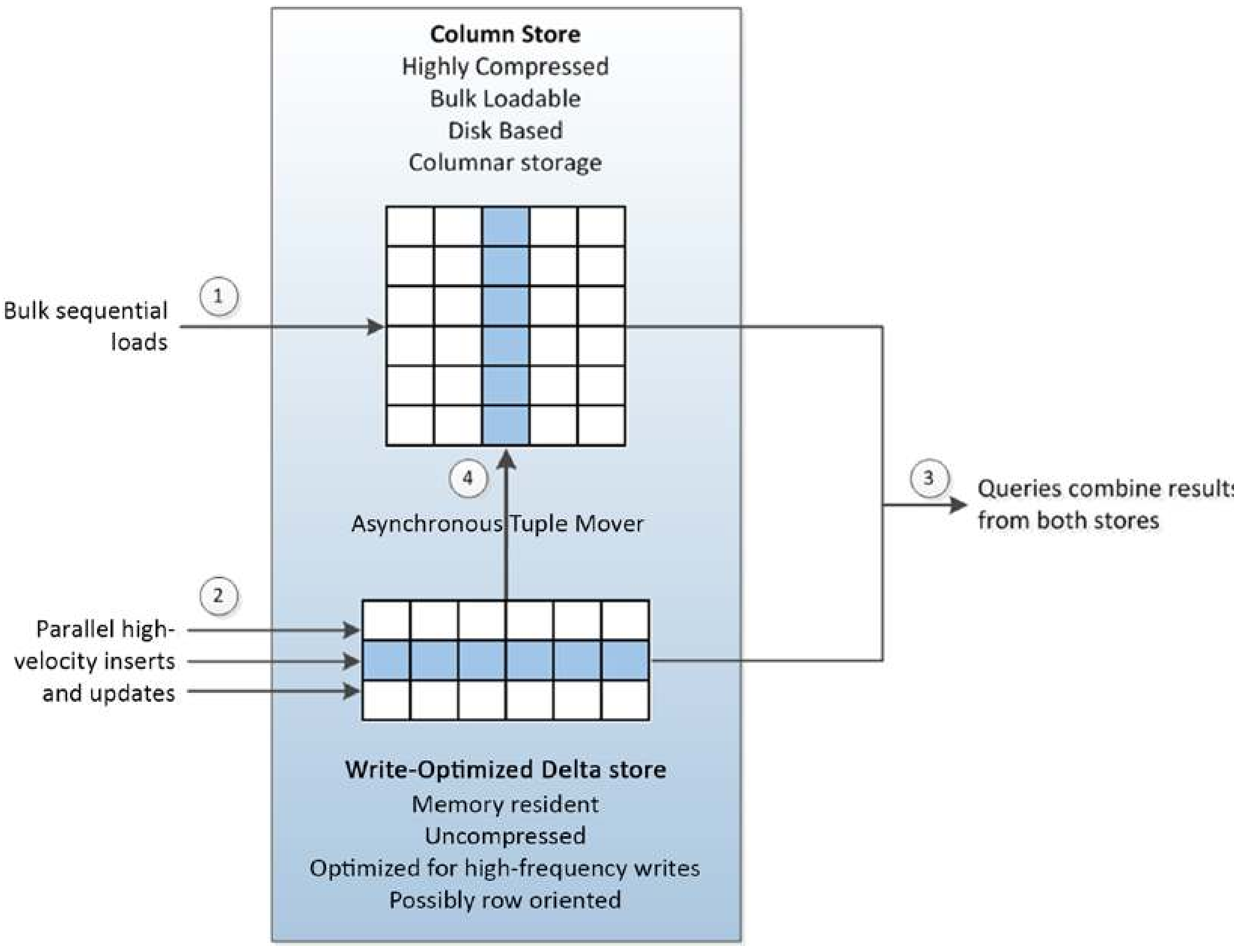
- Two main advantages
- Acceleration of aggregation queries
  - Because data are stored together
- Better data compression
  - Because similar data are stored together

Column database disadvantages



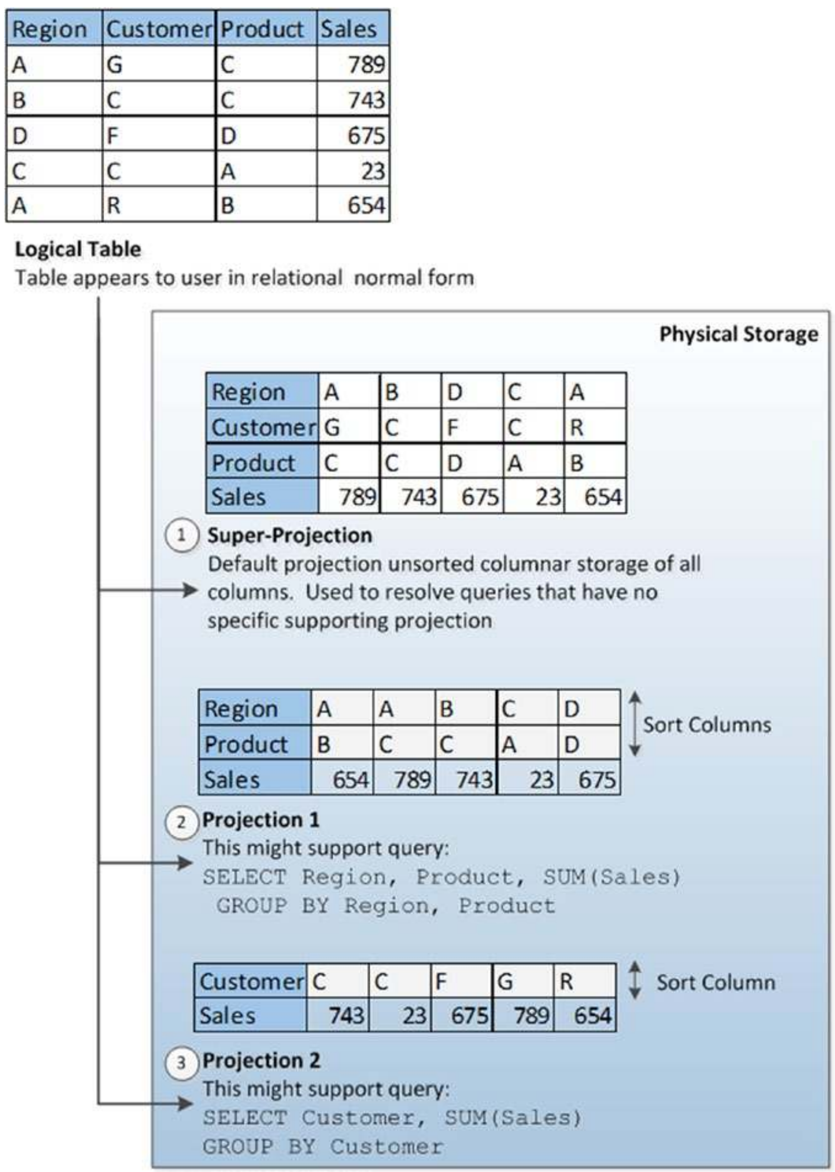
- Slower to insert
  - RDBMS requires one IO to insert a row
  - Column database could require one per column
- Inserts are generally batched across a number of rows

C-store is a column store with a row-based cache



- Column store is bulk-loaded periodically
- Delta-store is updated in real-time
- Queries combine results from both
- Company Vertica commercialized it

Data are stored as "Projections"



- Columns that are accessed together can be stored together
- Workload-dependent
  - Can be created manually
  - Or on the fly by the query optimizer
  - Or in bulk based on historical workloads
- Like indexes in RDBMS

Column store summary

- Designed for data warehousing and analytics
- Re-organize data for compression and aggregation
- Can be "added on" as feature to other DB systems
- Significant component in some in-memory DBs
  - For analytics applications (SAP HANA)
- For multimedia, store metadata in column store, media in key-value store

Column store
Strengths and weaknesses

- Strengths
  - Fast aggregations
  - Efficient compression
- Weaknesses
  - Vanilla model expensive to insert one row

Summary

- NoSQL architectures utilize different tradeoffs for different workloads
- Document databases for better fit with object-oriented code
- Graph databases for modeling relationships between things
- Column stores for efficient analytics