Database Systems

NoSQL

H. Turgut Uyar   Şule Öğüdücü

2005-2016

License

Topics

1. NoSQL
   - Introduction
   - Serialization: JSON
   - Key-Value Stores
   - Document Stores
2. XML Databases
   - Serialization: XML
3. Graph Databases

Topics

1. NoSQL
   - Introduction
   - Serialization: JSON
   - Key-Value Stores
   - Document Stores
2. XML Databases
   - Serialization: XML
3. Graph Databases

Relational Model

- relational model is not the best solution for all types of problems

- storing user preferences
- processing data from Wikipedia pages
- building a social network

Example: User Preferences

- user, preference type, selected option
- example task:
  retrieve notification setting of a given user
- no complex queries that would require SQL

Example: User Preferences

- user, preference type, selected option
- example task:
  retrieve notification setting of a given user
- no complex queries that would require SQL

Example: Wikipedia Pages

Casino Royale (2006 film)

From Wikipedia, the free encyclopedia

This article is about the 2006 film. For the 1967 film, see Casino Royale (1967 film). For other uses, see Casino Royale (disambiguation).

Casino Royale (2006) is the twenty-first film in the Eon Productions *James Bond* film series and the first to star Daniel Craig as the fictional MI6 agent James Bond. Directed by Martin Campbell and written by Neal Purvis & Robert Wade and Paul Haggis, the film marks the third screen adaptation of Ian Fleming's 1953 novel of the same name. *Casino Royale* is set at the beginning of Bond's career as Agent 007, just as he is earning his licence to kill. After preventing a terrorist attack at Miami International Airport, Bond falls in love with Vesper Lynd, the treasury employee assigned to provide the money he needs to bankrupt a terrorist financier, Le Chiffre, by beating him in a high-stakes poker game. The story arc continues in the following *Bond* film, *Quantum of Solace* (2008), with explicit references to characters and events in *Spectre* (2015).

Casino Royale reboots the series, establishing a new timeline and narrative framework not meant to precede or succeed any previous *Bond* film,[3][4] which allows the film to show a less experienced and more vulnerable Bond.[5] Additionally, the character Miss Moneypenny is, for the first time in the series, completely absent.[6] Casting the film involved a widespread search for a new actor to portray James Bond, and significant controversy surrounded Craig when he was selected to succeed Pierce Brosnan in October 2005. Location filming took place in the Czech Republic, the Bahamas, Italy and the United Kingdom with interior sets built at Pinewood Studios. Although part of the storyline is set in

Casino Royale

British cinema poster for *Casino Royale*, designed by Empire Design

Directed by	Martin Campbell
Produced by	Michael G. Wilson Barbara Broccoli
Screenplay by	Neal Purvis Robert Wade Paul Haggis
Based on	*Casino Royale* by Ian Fleming
Starring	Daniel Craig Eva Green Mads Mikkelsen Giancarlo Giannini Jeffrey Wright Judi Dench

- combination of structured and unstructured data
- example task:
  retrieve first paragraph
  of all James Bond movies
  starring Daniel Craig
- difficult to represent as a relation

Example: Wikipedia Pages

Casino Royale (2006 film)

From Wikipedia, the free encyclopedia

This article is about the 2006 film. For the 1967 film, see Casino Royale (1967 film). For other uses, see Casino Royale (disambiguation).

Casino Royale (2006) is the twenty-first film in the Eon Productions *James Bond* film series and the first to star Daniel Craig as the fictional MI6 agent James Bond. Directed by Martin Campbell and written by Neal Purvis & Robert Wade and Paul Haggis, the film marks the third screen adaptation of Ian Fleming's 1953 novel of the same name. *Casino Royale* is set at the beginning of Bond's career as Agent 007, just as he is earning his licence to kill. After preventing a terrorist attack at Miami International Airport, Bond falls in love with Vesper Lynd, the treasury employee assigned to provide the money he needs to bankrupt a terrorist financier, Le Chiffre, by beating him in a high-stakes poker game. The story arc continues in the following *Bond* film, *Quantum of Solace* (2008), with explicit references to characters and events in *Spectre* (2015).

Casino Royale reboots the series, establishing a new timeline and narrative framework not meant to precede or succeed any previous *Bond* film,[3][4] which allows the film to show a less experienced and more vulnerable Bond.[5] Additionally, the character Miss Moneypenny is, for the first time in the series, completely absent.[6] Casting the film involved a widespread search for a new actor to portray James Bond, and significant controversy surrounded Craig when he was selected to succeed Pierce Brosnan in October 2005. Location filming took place in the Czech Republic, the Bahamas, Italy and the United Kingdom with interior sets built at Pinewood Studios. Although part of the storyline is set in

Casino Royale

British cinema poster for *Casino Royale*, designed by Empire Design

Directed by	Martin Campbell
Produced by	Michael G. Wilson Barbara Broccoli
Screenplay by	Neal Purvis Robert Wade Paul Haggis
Based on	*Casino Royale* by Ian Fleming
Starring	Daniel Craig Eva Green Mads Mikkelsen Giancarlo Giannini Jeffrey Wright Judi Dench

- combination of structured and unstructured data
- example task:
  retrieve first paragraph
  of all James Bond movies
  starring Daniel Craig
- difficult to represent as a relation

Example: Social Network

- users: userid, name, age, gender, . . .
- friends: userid1, userid2
- example tasks:
  find all friends of a given user
  find all friends of friends of a given user
  find all female friends of male friends of a given user
  find all friends of friends of ... friends of a given user
- too many complicated joins

Example: Social Network

- users: userid, name, age, gender, . . .
- friends: userid1, userid2
- example tasks:
  find all friends of a given user
  find all friends of friends of a given user
  find all female friends of male friends of a given user
  find all friends of friends of … friends of a given user
- too many complicated joins

Problems: Representation

- difficult to handle unstructured and semistructured data
- difficult to represent hierarchy and neighborhood
- rigid schema: all rows need to store all fields
- even if not applicable
- fixed in advance
- to make changes: shut down, alter table, restart

Problems: Representation

- difficult to handle unstructured and semistructured data
- difficult to represent hierarchy and neighborhood

- rigid schema: all rows need to store all fields
- even if not applicable

- fixed in advance
- to make changes: shut down, alter table, restart

Problems: Scaling

- when volume of data increases:

- scale up: faster processor
- works up to a point

- scale out: more processors
- commodity hardware

Problems: Scaling

- when volume of data increases:
- scale up: faster processor
- works up to a point
- scale out: more processors
- commodity hardware

NoSQL

- NoSQL $\neq$ "don't use SQL"
- Not Only SQL
- use relational for some parts and non-relational for other parts
- key-value stores
- column family stores
- document stores
- graph databases

NoSQL

- NoSQL $\neq$ "don't use SQL"
- Not Only SQL
- use relational for some parts and non-relational for other parts

- key-value stores
- column family stores
- document stores
- graph databases

NoSQL Principles

- flexible schema
- focus on performance
- no joins
- massive scalability
- focus on availability
- updates should always be allowed

NoSQL Principles

- flexible schema
- focus on performance
- no joins
- massive scalability
- focus on availability
- updates should always be allowed

NoSQL Principles

- flexible schema
- focus on performance
- no joins
- massive scalability
- focus on availability
- updates should always be allowed

NoSQL Principles

- flexible schema
- focus on performance
- no joins
- massive scalability
- focus on availability
- updates should always be allowed

Availability vs Consistency

- focus on availability → relaxed consistency
- fewer transactional guarantees

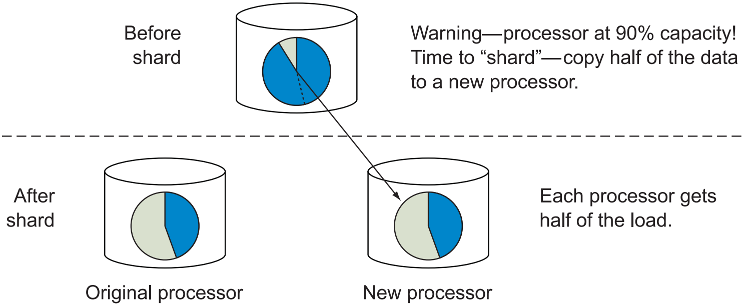
- BASE instead of ACID:
- Basic availability
- Soft state
- Eventual consistency

Sharding

- when a server nears full capacity for data
- sharding: break data into chunks
- spread chunks across distributed servers

- increases efficiency
- more servers $\rightarrow$ more points of failure

Sharding


Before
shard
Warning—processor at 90% capacity!
Time to “shard”—copy half of the data
to a new processor.
After
shard
Each processor gets
half of the load.
Original processor
New processor

Replication

- replicate data between servers
- increases fault tolerance

- copies might diverge
- eventual consistency: temporary inconsistency is allowed
- when system stops, all copies will be the same

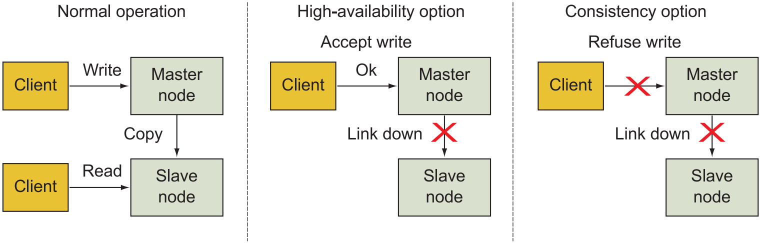
CAP Properties

- Consistency:
  all clients can read a single, up-to-date version of data from replicated partitions
- Availability:
  internal communication failures between replicated data don't prevent updates
- Partition tolerance:
  system keeps responding even if there is a communication failure between partitions

CAP Theorem

- Any distributed database can provide at most two of the three CAP properties. (Eric Brewer - 2000)

CAP


Normal operation
Client
Write
Master node
Copy
Client
Read
Slave node
High-availability option
Accept write
Client
Ok
Master node
Link down
Slave node
Consistency option
Refuse write
Client
Master node
Link down
Slave node

Serialization

- in which format can an object be stored?
- simple solution: string
- serial format

- when writing: object → serial format (serialization)
- when reading: serial format → object (deserialization)

Serialization

- in which format can an object be stored?
- simple solution: string
- serial format

- when writing: object $\rightarrow$ serial format (serialization)
- when reading: serial format $\rightarrow$ object (deserialization)

Serialization Formats

- common formats: XML, JSON
- human-readable
- useful for data interchange
- useful for representing semistructured data

Topics

1. NoSQL
   - Introduction
   - Serialization: JSON
   - Key-Value Stores
   - Document Stores
2. XML Databases
   - Serialization: XML
3. Graph Databases

JSON

- JavaScript Object Notation
- base values: number, string, ...
- objects: sets of key-value pairs
- arrays of values
- nested structure

JSON Example


```json
{
  "title": "The Usual Suspects",
  "year": 1995,
  "score": 8.7,
  "votes": 35027,
  "director": "Bryan Singer",
  "cast": [
    "Gabriel Byrne",
    "Benicio Del Toro"
  ]
}
```

JSON Example

```
[
  {
    "title": "The Usual Suspects",
    "year": 1995,
    ...
  },
  ...
  {
    "title": "Being John Malkovich",
    "year": 1999,
    ...
  }
]
```

Valid Documents

- JSON Schema

Query Language

- no commonly used, declarative query language
- programmatic handling of data

Topics

1. NoSQL
   - Introduction
   - Serialization: JSON
   - Key-Value Stores
   - Document Stores
2. XML Databases
   - Serialization: XML
3. Graph Databases

Key-Value Stores

- model: (key, value) pairs
- indexed by keys
- keys are distinct
- value is an arbitrary large blob of data
- very simple interface: put, get, delete
- no queries on values

- products: Redis, Riak, Memcache, Amazon DynamoDB

Key-Value Stores

- model: (key, value) pairs
- indexed by keys
- keys are distinct
- value is an arbitrary large blob of data
- very simple interface: put, get, delete
- no queries on values
- products: Redis, Riak, Memcache, Amazon DynamoDB

Key-Value Stores

- model: (key, value) pairs
- indexed by keys
- keys are distinct
- value is an arbitrary large blob of data

- very simple interface: put, get, delete
- no queries on values

- products: Redis, Riak, Memcache, Amazon DynamoDB

Key-Value Store Examples

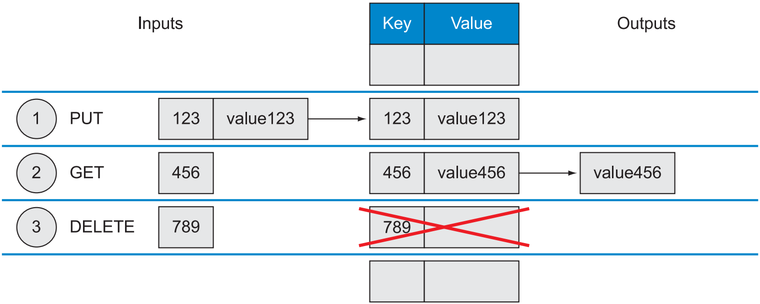
- web page caching
- key: URL, value: web page
- image store
- key: path to image, value: image

Key-Value Store Examples

- web page caching
- key: URL, value: web page

- image store
- key: path to image, value: image

Key-Value Store



Key-Value Stores

- distribute records to computing nodes based on key
- advanced: data structures in value
- not just a blob of data

Column Family Stores

- key is a (row, column) pair
- sparse matrix
- advanced keys: (row, column family, column, timestamp)
- column family: groups of columns
- timestamp: store multiple values over time
- products: Apache Cassandra, Apache HBase, Google BigTable

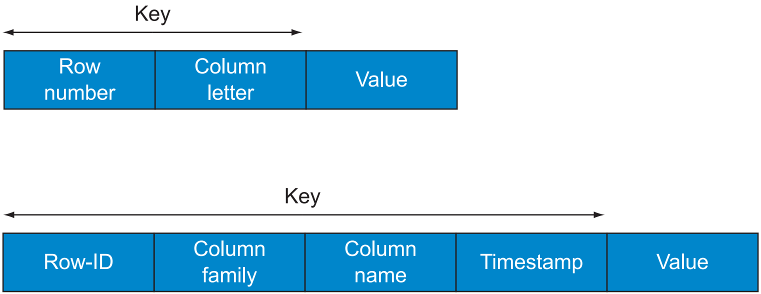
Column Family Stores

- key is a (row, column) pair
- sparse matrix

- advanced keys: (row, column family, column, timestamp)
- column family: groups of columns
- timestamp: store multiple values over time

- products: Apache Cassandra, Apache HBase, Google BigTable

Column Family Store



Column Family Store Example

- user preferences
- privacy settings, contact information, notifications, . . .
- typically under 100 fields, 1 KB

- only the associated user makes changes: no ACID requirements
- mostly read
- has to be fast and scalable

Topics

1. NoSQL
   - Introduction
   - Serialization: JSON
   - Key-Value Stores
   - Document Stores
2. XML Databases
   - Serialization: XML
3. Graph Databases

Document Stores

- model: (key, document) pairs
- document: JSON formatted data

- query based on document contents
- documents automatically indexed
- documents grouped into collections: hierarchical structure

- products: MongoDB, CouchDB
- application example: content management systems

MongoDB Insert Example

```
itucsdb.movies.insert(
  {
    "title": "Ed Wood",
    "year": 1994,
    "score": 7.8,
    "votes": 6587,
    "director": "Tim Burton",
    "cast": [
      "Johnny Depp"
    ]
  }
)
```

MongoDB Insert Example

```
itucsdb.movies.insert(
  {
    "title": "Three Kings",
    "year": 1999,
    "score": 7.7,
    "votes": 10319,
    "cast": [
      "George Clooney",
      "Spike Jonze"
    ]
  }
)
```

MongoDB Find Example

```
itucsdb.movies.find()

itucsdb.movies.find(
  {"year": 1999}
)

itucsdb.movies.find(
  {"year": {$gt 1999}}
)
```

Topics

XML

- XML is not a language itself
- framework for defining languages
- XML-based languages:
  XHTML, DocBook, SVG, ...
- XML processing languages:
  XPath, XQuery, XSL Transforms, ...

XML Structure

- an XML document forms a tree (hierarchy)
- nodes: *elements*
- elements represented by opening and closing tags
- nesting determines hierarchy
- non-container elements: self-closing tags
- elements can have attributes
- elements can have text as child node: character data (CDATA)

XML Example: XHTML

```
<html>
<head>
<title>...</title>
<meta charset="utf-8" />
</head>
<body>
<h1>...</h1>
<p>...</p>
<img src="..." alt="..." />
</body>
</html>
```

XML Example: Movie

```
<movie color="Color">
<title>The Usual Suspects</title>
<year>1995</year>
<score>8.7</score>
<votes>35027</votes>
<director>Bryan Singer</director>
<cast>
<actor>Gabriel Byrne</actor>
<actor>Benicio Del Toro</actor>
</cast>
</movie>
```

XML Example: Movies

```
<movies>
<movie color="Color">
<title>The Usual Suspects</title>
<year>1995</year>
...
</movie>
<movie color="Color">
<title>Being John Malkovich</title>
<year>1999</year>
...
</movie>
...
</movies>
```

Well-Formed Documents

- well-formed: conforming to XML rules
- syntactically correct

- single root element
- proper nesting of elements: matched tags
- unique attributes within elements

- XML parsers convert well-formed XML documents into DOM objects (Document Object Model)

Well-Formed Documents

- well-formed: conforming to XML rules
- syntactically correct

- single root element
- proper nesting of elements: matched tags
- unique attributes within elements

- XML parsers convert well-formed XML documents into DOM objects (Document Object Model)

Valid Documents

- valid: conforming to domain rules
- semantically correct
- DTD, XML Schema
- validating XML parsers also check for validity

Valid Documents

- valid: conforming to domain rules
- semantically correct
- DTD, XML Schema
- validating XML parsers also check for validity

XML Databases

- variant of document stores
- document: XML formatted data
- query using XPath
- products: Oracle Berkeley DBXML, BaseX, eXist

XPath

- XPath: selecting nodes and data from XML documents

- path of nodes to find: chain of location steps
- starting from the root (absolute)
- starting from the current node (relative)

XPath Examples

- all movies: `/movies/movie`
- actors of current movie: `./cast/actor`
- `../../year`

Location Steps

- location step structure:
  `axis::node_selector[predicate]`
- axis: where to search
- selector: what to search
- predicate: under which conditions

Axes

- `child`: all children, one level (default axis)
- `descendant`: all children, recursively (shorthand: `//`)
- `parent`: parent node, one level
- `ancestor`: parent nodes, up to document element
- `attribute`: attributes (shorthand: `@`)
- `following-sibling`: siblings that come later
- `preceding-sibling`: siblings that come earlier
- ...

Node Selectors

- node tag
- node attribute
- node text: `text()`
- all children: $*$

XPath Examples

- names of all directors:
  `/movies/movie/director/text()`
  `//director/text()`
- all actors in this movie:
  `./cast/actor`
  `.//actor`
- colors of all movies:
  `//movie/@color`
- scores of movies after this one:
  `./following-sibling::movie/score`

XPath Examples

- names of all directors:
  `/movies/movie/director/text()`
  `//director/text()`
- all actors in this movie:
  `./cast/actor`
  `.//actor`
- colors of all movies:
  `//movie/@color`
- scores of movies after this one:
  `./following-sibling::movie/score`

XPath Examples

- names of all directors:
  `/movies/movie/director/text()`
  `//director/text()`
- all actors in this movie:
  `./cast/actor`
  `.//actor`
- colors of all movies:
  `//movie/@color`
- scores of movies after this one:
  `./following-sibling::movie/score`

XPath Examples

- names of all directors:
  `/movies/movie/director/text()`
  `//director/text()`
- all actors in this movie:
  `./cast/actor`
  `.//actor`
- colors of all movies:
  `//movie/@color`
- scores of movies after this one:
  `./following-sibling::movie/score`

XPath Predicates

- testing node position: `[position]`
- testing existence of a child: `[child_tag]`
- testing value of a child: `[child_tag="value"]`
- testing existence of an attribute: `[@attribute]`
- testing value of an attribute: `[@attribute="value"]`

XPath Predicates

- testing node position: `[position]`
- testing existence of a child: `[child_tag]`
- testing value of a child: `[child_tag="value"]`
- testing existence of an attribute: `[@attribute]`
- testing value of an attribute: `[@attribute="value"]`

XPath Predicates

- testing node position: `[position]`
- testing existence of a child: `[child_tag]`
- testing value of a child: `[child_tag="value"]`
- testing existence of an attribute: `[@attribute]`
- testing value of an attribute: `[@attribute="value"]`

XPath Examples

- title of the first movie:
  `/movies/movie[1]/title`
- all movies in the year 1997:
  `movie[year="1997"]`
- black-and-white movies:
  `movie[@color="BW"]`

XPath Examples

- title of the first movie:
  `/movies/movie[1]/title`
- all movies in the year 1997:
  `movie[year="1997"]`
- black-and-white movies:
  `movie[@color="BW"]`

XPath Examples

- title of the first movie:
  `/movies/movie[1]/title`
- all movies in the year 1997:
  `movie[year="1997"]`
- black-and-white movies:
  `movie[@color="BW"]`

Graph Databases

- model: nodes and edges
- nodes have properties
- edges have labels

- for relationship intensive data: social networks, ...
- traversals instead of joins

- products: Neo4J

Graph Databases

- better suited for tasks like:
  shortest path, friends of friends,
  neighboring nodes with specific properties
- difficult to scale out
- declarative query languages: Cypher, Gremlin

Graph Databases

- better suited for tasks like:
  shortest path, friends of friends,
  neighboring nodes with specific properties
- difficult to scale out
- declarative query languages: Cypher, Gremlin

Graph Databases

- better suited for tasks like:
  shortest path, friends of friends,
  neighboring nodes with specific properties
- difficult to scale out
- declarative query languages: Cypher, Gremlin

Cypher

- locate the initial nodes
- select and traverse relationships
- change and/or return values

Cypher: Nodes

- `(name)`
- `(name:Type)`
- `(name:Type {attributes})`

```
(matrix)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

Cypher: Relationships

- undirected: `--`
- directed: `-->` `<--`
- with details: `-[]-`

```
-[role]->
-[role:ACTED_IN]->
-[role:ACTED_IN {roles: ["Neo"]}]->
```

Cypher: Patterns

- combine nodes and relationships
- give names to patterns

```
(keanu:Person   {name:  "Keanu Reeves"} )
-[role:ACTED_IN {roles: ["Neo"] } ]->
(matrix:Movie   {title: "The Matrix"} )

acted_in = (:Person)-[:ACTED_IN]->(:Movie)
```

Cypher: Creating Data

```
CREATE (:Movie {title: "The Matrix", released: 1997})

CREATE (p:Person {name: "Keanu Reeves", born: 1964})
RETURN p

CREATE (a:Person {name: "Tom Hanks", born:1956 })
  -[r:ACTED_IN {roles: ["Forrest"]}]->
  (m:Movie {title: "Forrest Gump", released: 1994})
CREATE (d:Person {name: "Robert Zemeckis", born: 1951})
  -[:DIRECTED]-> (m)
RETURN a, d, r, m
```

Cypher: Matching Patterns

```
MATCH (m:Movie)
RETURN m

MATCH (p:Person {name:"Keanu Reeves"})
RETURN p

MATCH (p:Person {name:"Tom Hanks"})
  -[r:ACTED_IN]-> (m:Movie)
RETURN m.title, r.roles
```

References

Supplementary Reading

- Making Sense of NoSQL, by Dan McCreary and Ann Kelly, Manning Publications
- The Neo4J Manual: Tutorials
  `http://neo4j.com/docs/stable/tutorials.html`