Advanced Management of Data

NotOnlySQL Systems (1)

Semistructured Data

Structured Data (in relational / object-relational DBMSs)

- each record follows the same format as other records

Motivation

- sometimes similar data objects must be described differently
- often data is (differently) collected before it is known how it will be stored and managed
- different web sources as data sources (database) cannot be constrained with a schema
- it may be desirable to have a flexible format for data exchange between disparate databases

Semistructured Data (schema-less / self-describing data)

- data may have a certain structure, but not necessarily an identical structure
- some attributes may be shared, but some other attributes may be used by only a few entities
- there is no predefined schema, instead the schema information is mixed in with the data values

Displaying semistructured data

- directed graph
- labels of edges represent schema names (attribute names, object / entity types, classes, relationships)
- internal nodes represent individual objects or composite attributes
- leaf nodes represent actual data values of atomic attributes

PROJECT

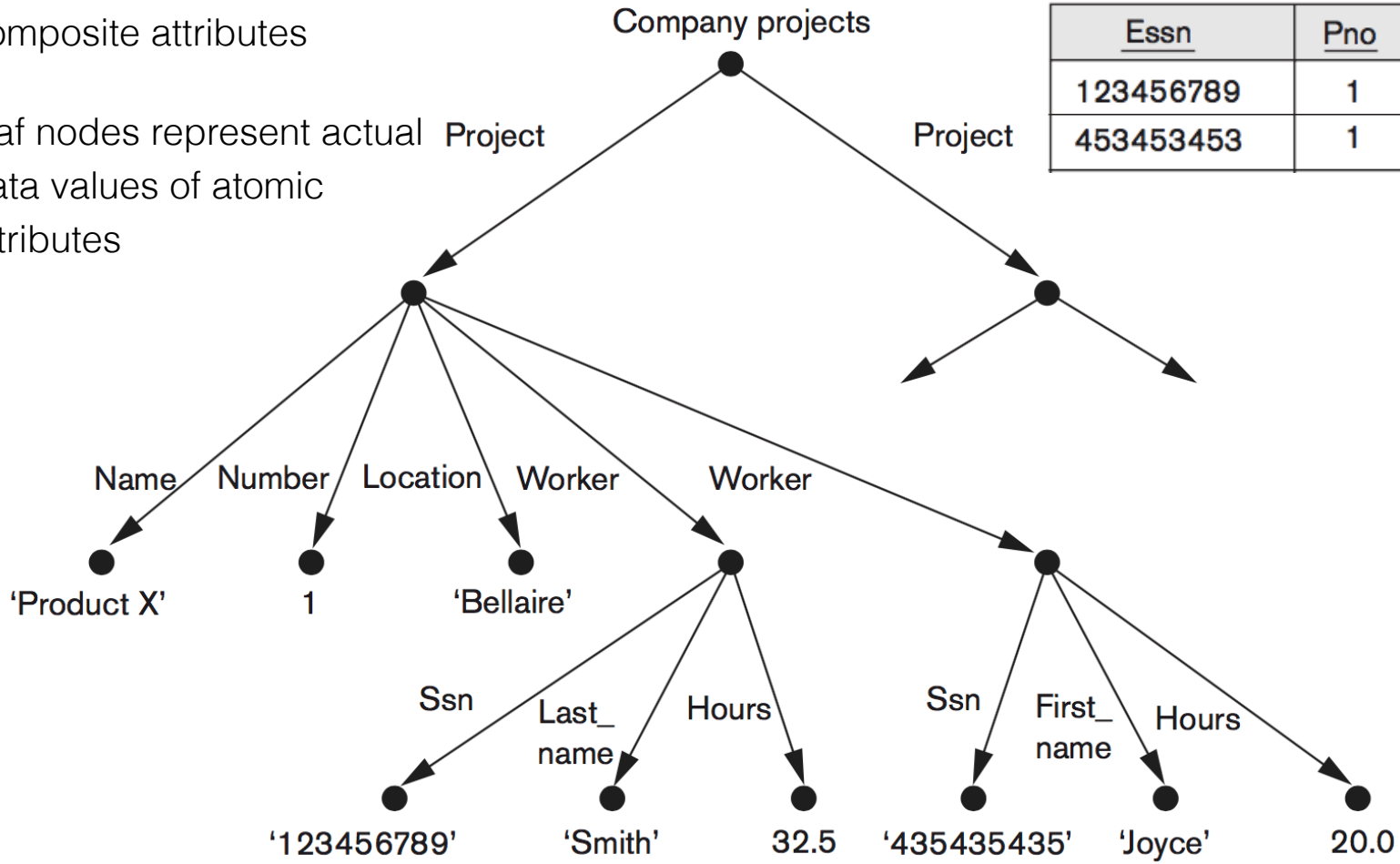
Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5

EMPD_5

Fname	Minit	Lname	Ssn
John	B	Smith	123456789
Joyce	A	English	453453453

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
453453453	1	20.0



[Elmasri & Navathe]

Semistructured Data

Some languages to describe semistructured data

XML (Extended Markup Language)

JSON (Javascript Object Notation)

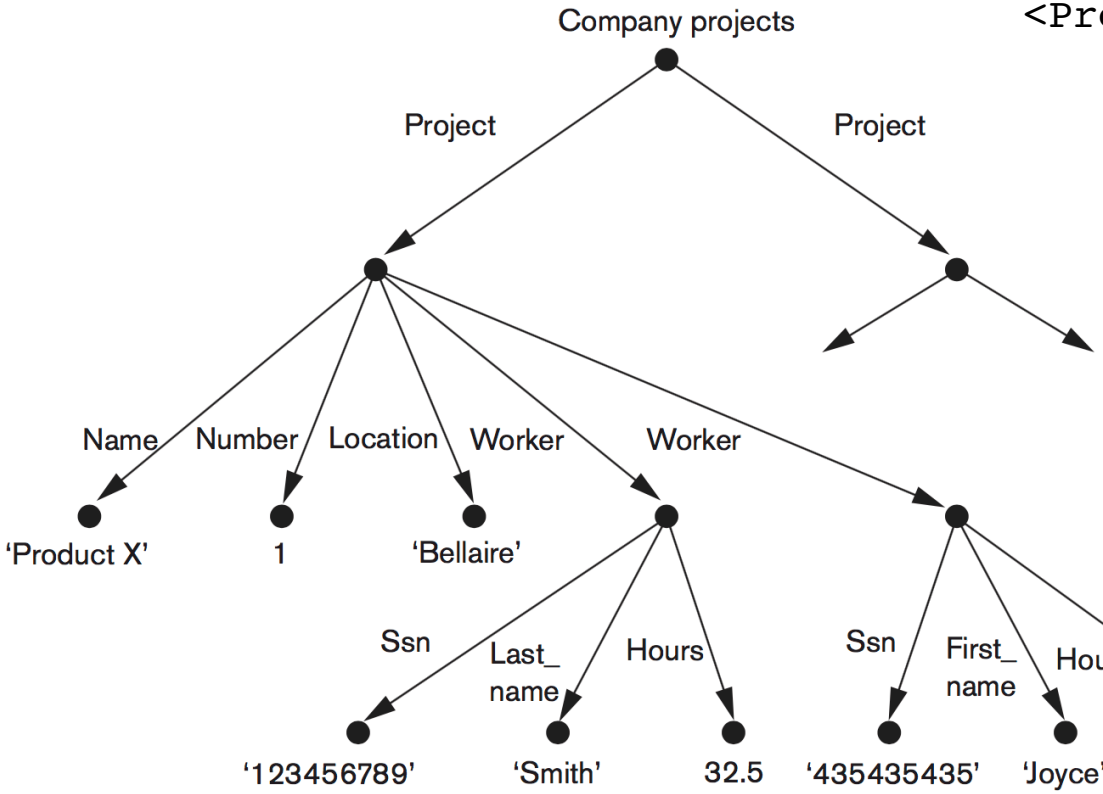
Querying semistructured data

Most of the approaches to semistructured data management are based on query languages that traverse a tree-labeled representation, e.g. XPath and XQuery for XML

Without a schema, data can be identified only by specifying its position within the collection.

➔ Data items are now accessed navigational rather than by declarative descriptions, which are based on structural properties.

Semistructured Data - XML




```
<?xml version="1.0" standalone="yes"?>
<Projects>
    <Project>
        <Name>ProductX</Name>
        <Number>1</Number>
        <Location>Bellaire</Location>
        <Dept_no>5</Dept_no>
        <Worker>
            <Ssn>123456789</Ssn>
            <Last_name>Smith</Last_name>
            <Hours>32.5</Hours>
        </Worker>
        <Worker>
            <Ssn>453453453</Ssn>
            <First_name>Joyce</First_name>
            <Hours>20.0</Hours>
        </Worker>
    </Project>
    <Project>
    …
    </Project>
</Projects>
```


[Elmasri & Navathe]

NOSQL Systems - Introduction

Focus of traditional relational systems

- centralized data storage
- structured data storage
- data consistency / ACID compliance
  (a transaction must be **a**tomic, **c**onsistent, **i**solated, and **d**urable)
- aggregation
- powerful query languages

Focus of NOSQL systems

- distributed data storage
- flexible data storage
  - schema-less data sets that include structured and semistructured data
- scalability
- high performance (at scale)
- availability through replication

The CAP Theorem

The CAP theorem

In a distributed system with data replication only two of the following properties can be guaranteed at the same time:

- **C**onsistency — The nodes will have the same copies of a replicated data item visible for various transactions.
- **A**vailability — Each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.
- **P**artition tolerance — The system can continue to operate if network partitioning happens

NOSQL characteristics

Focus of traditional SQL applications

- keeping serializable consistency (strongest form of consistency) through the ACID properties of a transaction
- implies that write performance can slow down because an update must be applied to every copy of the replicated data items

Focus of NOSQL applications

- guaranteeing availability and partition tolerance ➔ only weaker forms of consistency can be used

Eventual Consistency

If no new updates are made to a given data item eventual consistency informally guarantees that eventually all accesses to that item will return the last updated value.

NOSQL characteristics

Horizontal Scalability (Scale out)

- NOSQL systems are designed to scale horizontally by adding more nodes for data storage and processing as the volume of data grows
- NOSQL systems don't require expensive and specialized storage and processing hardware but use low cost commodity servers
- more nodes can be added while the system is operational

  ➔ techniques for distributing the existing data among new nodes without interrupting system operation are necessary

NOSQL characteristics

Replication Model: Master-slave replication

- requires one copy to be the master copy
- all write operations must be applied to the master copy and then propagated to the slave copies by using eventual consistency
- for read operations, the master-slave technique can be configured:
  - all read operations access the master copy
    - this is similar to the primary site / copy methods of distributed concurrency control
  - read operations can access the slave copies
    - it is not guaranteed that the read values are the latest written values, since write operations to the slave nodes can be done after they are applied to the master copy

NOSQL characteristics

Replication Model: Master-master replication

- allows reads and writes at any of the replicas
- may not guarantee that read operations at nodes that store different copies see the same values
- different users may write the same data item concurrently at different nodes of the system
  ➔ the values of the item will be temporarily inconsistent.

  ➔ a method to resolve conflicting write operations of the same data item at different nodes must be implemented

NOSQL characteristics

Sharding

Often, data collections of NOSQL applications can have many billions of records.

This vast amount of data may be accessed by millions of users concurrently.

So it is not possible (or practical) to store all records in one node.

Sharding (horizontal partitioning) is often employed in NOSQL systems to distribute the records to multiple nodes.

Combining sharding and replicating (the shards) improves both

- load balancing
- data availability

NOSQL characteristics

High-Performance Data Access

To quickly find an individual data item among millions or billions of records, data access is realized based only on key values (object key) rather than using complex query conditions.

Optionally, other indexes can be created to locate data items based on attribute conditions.

Most systems use one of two techniques: hashing or range partitioning on object keys:

Hashing

A hash function *h(K)* is applied to the key *K*, and the location of the object with key *K* is determined by the value of *h(K)* ➔ *O(1).*

Range partitioning

Data location is determined via a range of key values, which is adequate for applications that require range queries.

NOSQL characteristics

(No) Schema

- most NOSQL systems do not require to have a schema
- instead semi-structured, self-describing data is used
- users can specify partial schemas in some systems to improve storage efficiency

➔ External constraint definition

- without a schema, any constraints on the data have to be programmed in the application programs that access the data items

NOSQL characteristics

Less Powerful Query Languages

- many NOSQL applications do (and can) not require a powerful query language such as SQL
- search queries often only locate single objects in a single file based on their object keys
- NOSQL systems typically provide a set of basic functions as a programming API, which are called CRUD operations, for Create, Read, Update, and Delete
- some NOSQL systems provide a high-level query language, but in most cases only a subset of SQL querying capabilities is provided
- many NOSQL systems do not provide join operations as part of the query language itself
  ➔ the joins need to be implemented in the application programs

NOSQL Systems

Categories of NOSQL Systems

- Document stores
- Key-value stores
- Wide column stores
- Graph-Databases
- Hybrid Systems

Document Stores

Document-based NOSQL systems store data as collections of similar documents.

Documents

- resemble complex objects
- do not require to specify a schema, but are specified as self-describing data
- each document can have different data elements (attributes)
- can be specified in various formats, such as XML or JSON
- are accessible via their document id
- can also be accessed using other indexes, which are created by the system by extracting the data element names (on user's request)

MongoDB

MongoDB Data Model

- documents are stored in BSON (Binary JSON) format
- collections do not have a schema
- the structure of the data fields depends on access methods and usage of documents:
  - normalized design, which is similar to normalized relational tuples
  - denormalized design, which is similar to XML documents or complex objects
- interdocument references can be specified by storing the document id(s) of related document(s)

Example

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5

EMPD_5

Fname	Minit	Lname	Ssn
John	B	Smith	123456789
Joyce	A	English	453453453

[Elmasri & Navathe]

Creating collections

- to store PROJECT objects from the COMPANY database:

```
db.createCollection("project",{capped:true, size:1310720, max:500})
```

- to hold information about the EMPLOYEEs who work on each project:

```
db.createCollection("worker",{capped:true, size:5242880, max:2000})
```

Capping-Parameter:

Helps the system to choose the storage options for a collection: **`capped:true`** means, that a collection has upper limits on its storage space (**`size`**) and number of documents (**`max`**)

ObjectID

_id

Each document in a collection has an ObjectID field, which is automatically indexed.

User-generated ObjectID:

- can have any value specified by the user as long as it uniquely identifies the document

System-generated ObjectID (if the user does not specify an _id field):

- is a 16-byte id value, which combines
  - the timestamp when the object is created (4 bytes)
  - the node id (3 bytes)
  - the process id (2 bytes)
  - a counter (3 bytes)

Example

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5

EMPD_5

Fname	Minit	Lname	Ssn
John	B	Smith	123456789
Joyce	A	English	453453453

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
453453453	1	20.0

[Elmasri & Navathe]

project document with an array of embedded workers:

```
{
    _id:        "P1",
    Pname:      "ProductX",
    Plocation: "Bellaire",
    Workers: [
                {
                  Ename: "John Smith",
                  Hours: 32.5
                },
                {
                    Ename: "Joyce English“,
                    Hours: 20.0
                }
            ]
};
```

- _id values are user-defined here
- a list of values that is enclosed in square brackets [...] represents a field whose value is an array

Example

„project“ document with an embedded array of worker ids:

```
{
    _id:        "P1",
    Pname:      "ProductX",
    Plocation: "Bellaire",
    WorkerIds: [ "W1", "W2" ]
}
```

„worker“ collection:

```
{
    _id:    "W1",
    Ename: "John Smith",
    Hours: 32.5
}

{
    _id:    "W2",
    Ename: "Joyce English",
    Hours: 20.0
}
```

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5

EMPD_5

Fname	Minit	Lname	Ssn
John	B	Smith	123456789
Joyce	A	English	453453453

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
453453453	1	20.0

[Elmasri & Navathe]

Example

„worker“ and „project“ documents:

```
{
    _id:        "W1",
    Ename:      "John Smith",
    ProjectId:  "P1",
    Hours:      32.5
}

{
    _id:        "W2",
    Ename:      "Joyce English",
    ProjectId:  "P1",
    Hours:      20.0
}
```

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5

EMPD_5

Fname	Minit	Lname	Ssn
John	B	Smith	123456789
Joyce	A	English	453453453

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
453453453	1	20.0

[Elmasri & Navathe]

```
{
    _id:        "P1",
    Pname:      "ProductX",
    Plocation:  "Bellaire"
}
```

MongoDB - CRUD Operations

Create

```
db.<collection_name>.insert(<document(s)>)
```

Read

```
db.<collection_name>.find(<condition>)
```

Update

```
db.<collection_name>.update(…)
```

Modifies existing document(s) in a collection. Specific fields of existing document(s) can be specified or existing document(s) can be replaced entirely.

Delete

```
db.<collection_name>.remove(<condition>)
```

<condition>

- specifies a general boolean condition
- the documents in the collection that return `true` are selected for the operation result

Example

Example

- inserting the documents into their collections “project” and “worker”:

```
db.project.insert(
    {
        _id:       "P1",
        Pname:     "ProductX",
        Plocation: "Bellaire"
    }
)

db.worker.insert(
    [
        {
            _id:        "W1",
            Ename:      "John Smith",
            ProjectId: "P1",
            Hours:      32.5
        },
        {
            _id:        "W2",
            Ename:      "Joyce English",
            ProjectId: "P1",
            Hours:      20.0
        }
    ]
)
```

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5

EMPD_5

Fname	Minit	Lname	Ssn
John	B	Smith	123456789
Joyce	A	English	453453453

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
453453453	1	20.0

[Elmasri & Navathe]

MongoDB

Replication

MongoDB uses a variation of the master-slave approach for replication:

Multiple copies (replica set) of the same data set are created on different nodes.

All write operations must be applied to the primary copy and then propagated to the secondaries.

For read operations, the user can choose the particular read preference for their application:

- The default read preference processes all read operations at the primary copy, so all reads and writes are performed at the primary node.
- An optional read preference enables processing of read operations at primary or secondary replicas. This method increases read performance, but a read at a secondary replica does not guarantee to get the latest version of a document.

MongoDB

Sharding

- divides the documents into disjoint partitions

Shard key

Basis for partitioning is a user specified document field that must have two characteristics:

- it must exist in every document in the collection
- it must have an index

Chunks

Documents are partitioned based on the division of the values of the shard key.

- Range partitioning creates the chunks by specifying a range of key values
- Hash partitioning applies a hash function *h(K)* to each shard key *K*, and the partitioning of keys into chunks is based on the hash values

MongoDB

Query Router

- is a MongoDB module that keeps track of which nodes contain which shards
- a query (CRUD operation) will be routed to the nodes that contain the shards that hold the requested documents
- if the system cannot determine which shards hold the required documents, the query will be submitted to all the nodes that hold shards of the collection
- sharding and replication are used together
    - sharding focuses on improving performance via load balancing and horizontal scalability
    - replication focuses on ensuring system availability when certain nodes fail in the distributed system