

282

MongoDB

- ドキュメント指向NoSQLデータベースの一つ
- Mongoという名前は、英語で「ばかでかい」を意味する "humongous" に由来する
- JSONを拡張したBSONの形式でデータを格納する
- データの構造をあらかじめ定義しておく必要がない(スキーマレス)
- 任意の比較演算子での結合問合せはない(\$lookupを使った等号での結合のみ)

283

リレーショナルデータベースとの対応

| リレーショナルデータベース | MongoDB |
|---------------|-----------|
| データベース | データベース |
| 表 | コレクション |
| 行 | ドキュメント |
| 列 | フィールド |
| タプル | オブジェクト |
| 主キー | _id フィールド |

284

JSONフォーマット

- JavaScript Object Notationの略
- オブジェクト(フィールド名と値のペアの集まり)でデータを表現する

(例) {empno: "E01", ename: "Smith"}
{フィールド名1: 値1, フィールド名2: 値2, ..., フィールド名n: 値n}

- オブジェクトの入れ子構造(オブジェクトの値のところに新たなオブジェクトを書いたもの)を扱える
- オブジェクトの値として配列(データの系列)を扱うことができる

(例) { goods: ["milk", "bread", "butter"] }

285

データの登録

db.コレクション名.insertOne(ドキュメント)

- 1つのドキュメントをコレクションに登録する
- SQLの insert into 表 values タプル に相当(注意)SQLと違ってスキーマの定義が必要ない(コレクションや中身のドキュメントは自動的に生成される)

db.コレクション名.insertMany([ドキュメント1, ドキュメント2, ..., ドキュメントn])

- 複数のドキュメントをコレクションに登録する(ドキュメントの系列を配列で表現)

286

データの登録例(1)

```
db.emp.insertOne( {empno: "E01",
ename: "Smith", deptno: "D01", salary: 100} )
```

(参考)SQLの insert into emp values ('E01', 'Smith', 'D01', 100) に相当

内部での表現(_idは自動的に作成されるフィールド(RDBMSの主キーに対応))

```
{ "_id" : ObjectId("587ceffb3ae75d7ed54218de"),
"empno" : "E01", "ename" : "Smith", "deptno" : "D01",
"salary" : 100 }
```

287

実習で使うデータベース(一部)

```
db.emp.insertMany( [
{ empno: "E01", ename: "Smith", deptno: "D01",
salary: 100 },
{ empno: "E02", ename: "Morgan", deptno: "D01",
salary: 70 },
{ empno: "E03", ename: "Robert", deptno: "D01",
salary: 80 },
{ empno: "E04", ename: "Washington", deptno: "D02",
salary: 120 },
.....
] )
```

288

実習で使うデータベース(つづき)

```

db.dept.insertMany( [
  { deptno: "D01", dname: "Account", manager: "E01" },
  { deptno: "D02", dname: "Personnel", manager: "E04" },
  { deptno: "D03", dname: "Education", manager: "E07" },
  { deptno: "D91", dname: "Database", manager: "E91" }
] )
    
```

289

データベースの問合せ

db.コレクション名.find(検索条件, 出力フィールド指定)

- 検索条件と出力フィールド指定はオブジェクトの形式で表現する(どちらも省略可能で省略すると条件なしとなる)
- 検索条件の表現

{フィールド名: 値} ← 値をそのまま書くと等号を指定したことになる(フィールド名 = 値)

{フィールド名: {比較演算子: 値}} ← フィールドの値と比較演算子で比較(\$gt (>), \$gte (>=), \$lt (<), \$lte (<=), \$ne (≠) などがある)

{{フィールド名1: 値1}, {フィールド名2: 値2}} ← 並べて書くことANDになる(フィールド名1=値1 and フィールド名2=値2)

290

検索条件

- > {フィールド名: 値} ← 値をそのまま書くと等号を指定したことになる(フィールド名 = 値)
- > {フィールド名: {比較演算子: 値}} ← フィールドの値と比較演算子で比較(\$gt (>), \$gte (>=), \$lt (<), \$lte (<=), \$ne (≠) などがある)
- > {フィールド名1: 値1, フィールド名2: 値2} ← 並べて書くことANDになる(フィールド名1=値1 and フィールド名2=値2)
- > { \$or: [{フィールド名1: 値1}, {フィールド名2: 値2}] } ← 条件のORを指定するには\$orの後に配列で表現する(フィールド名1=値1 or フィールド名2=値2)
- > \$orのところで\$andと書いて条件のANDの指定も可能

291

検索条件の例

db.emp.find({ ename: "Smith" }) ← ename = "Smith" に相当

db.emp.find({ salary: { \$gte: 100 } }) ← salary >= 100に相当

db.emp.find({ deptno: "D01", salary: { \$lt: 100 } }) ← deptno="D01" and salary<100に相当

db.emp.find({ \$or: [{deptno: "D03"}, {salary: { \$gt: 120 } }] }) ← deptno="D03" or salary>120に相当

292

出力フィールド指定

{フィールド名: 数値} 値が0なら指定したフィールドを出力しない、0以外なら出力する

例 { deptno: 0, dname: 1 } dnameは出力するが、deptnoは出力しない(出力フィールド指定をつけると通常のフィールドは出力しなくなる)

(注意) _idフィールドは明示的に0を指定しないと常に出力される

実行例

```

db.dept.find( {}, { dname: 1 } )
{ "_id": ObjectId("5c07a660427bcb83dd6bb58"), "dname": "Account" }
{ "_id": ObjectId("5c07a660427bcb83dd6bb59"), "dname": "Personnel" }
...
db.dept.find( {}, { _id: 0, dname: 1 } )
{ "dname": "Account" }
{ "dname": "Personnel" }
...
    
```

293

SQLとの比較(1)

| SQL | MongoDB |
|---|---|
| select * from emp | db.emp.find() |
| select * from emp where empno='E01' | db.emp.find({empno: "E01"}) <small>等号の比較は値をそのまま書く</small> |
| select ename from emp where empno='E01' | db.emp.find({empno: "E01"}, {ename: 1}) <small>第2引数は出力するフィールドの指定(値0は出力しない、0以外だと出力)</small> |

294

SQLとの比較(2)

| SQL | MongoDB |
|--|---|
| <pre>select * from emp where salary >= 90 and salary <= 120</pre> | <pre>db.emp.find({salary : {\$gte: 90, \$lte: 120}}) <small>比較演算子には\$gt (>), \$gte (>=), \$lt (<), \$lte (<=), \$ne (!)などがある</small></pre> |
| <pre>select emp.*, dept.* from emp join dept on emp.deptno = dept.deptno</pre> | <pre>db.emp.aggregate([{ \$lookup: {from:"dept", localField:"deptno", foreignField: "deptno", as: "emp_dept"} }]) <small>結合問合せに相当する</small></pre> |

295

集約パイプライン (aggregate pipeline)

- 複数の問合せ処理を組み合わせて実行する
- 個々の処理をステージと呼び、前段のステージの結果が後段のステージに渡される
- ステージは配列の要素として表現する

```
db.コレクション名.aggregate(
[ {ステージ1}, {ステージ2}, ..., {ステージn} ]
)
```

296

集約パイプラインのステージ例

- \$match 検索条件を満たすドキュメントを選択する (findの検索条件と同じ)
- \$lookup 他のコレクションのドキュメントと結合する (SQLのjoinに相当する)
- \$group 問合せの結果をグループ化する
- \$project 出力のフィールドを指定する (findの出力フィールド指定と同じ)

297

\$matchステージ

```
{ $match: { 検索条件 } }
```

検索条件を満たすドキュメントを選択する (findの検索条件と同じ)

```
db.emp.aggregate( [
{ $match: {ename: "Smith"} }
] )
```

結果

```
{ "_id" : ObjectId("5c07a660427bcba83dd6bb4b"),
"empno" : "E01", "ename" : "Smith", "deptno" :
"D01", "salary" : 100 }
```

298

\$lookupステージ

他のコレクションのドキュメントと結合する (SQLのjoinに相当する)

```
{ $lookup: {
from: "結合先のコレクション名",
localField: "結合対象のフィールド名 (結合元)",
foreignField: "結合対象のフィールド名 (結合先)",
as: "結合で新たにできるフィールドの名前"
} }
```

299

\$lookupの使用例

```
db.emp.aggregate([
{ $match: {ename: "Smith"} },
{ $lookup: { from:"dept", localField:"deptno",
foreignField:"deptno", as: "emp_dept" } }
] )
```

結果

```
{ "_id" : ObjectId("5c07a660427bcba83dd6bb4b"),
"empno" : "E01", "ename" : "Smith", "deptno" : "D01",
"salary" : 100, "emp_dept" : [ { "_id" :
ObjectId("5c07a660427bcba83dd6bb58"), "deptno" :
"D01", "dname" : "Account", "manager" : "E01" } ] }
```

300

\$groupステージ

問合せの結果をグループ化する(SQLのgroup byに相当する)

```
{ $group: { _id: "$グループ化するフィールド名",
           グループ化で作るフィールド名:
             { 集約演算子: 集約対象 } } }
```

(注意)全体で集約するときは_id: null とする
集約演算子には以下のようなものがある(一部)
\$max(最大値)、\$min(最小値)、\$avg(平均)、
\$sum(合計)、\$push(フィールドの値を集約)

301

\$groupの使用例(1)

```
db.emp.aggregate([
  { $group: { _id: "$deptno", count: { $sum: 1 } } }
])
```

結果

```
{ "_id": null, "count": 1 }
{ "_id": "D01", "count": 4 }
{ "_id": "D03", "count": 3 }
{ "_id": "D02", "count": 2 }
{ "_id": "D91", "count": 3 }
```

```
db.emp.aggregate([ { $group: { _id: null, count: { $sum: 1 } } } ])
```

結果

```
{ "_id": null, "count": 13 }
```

302

\$groupの使用例(2)

```
db.emp.aggregate([
  { $group: { _id: "$deptno",
             ename: { $push: "$ename" } } }
])
```

結果

```
{ "_id": null, "ename": [ "Roosevelt" ] }
{ "_id": "D01", "ename": [ "Smith", "Morgan", "Robert", "Taylor" ] }
{ "_id": "D03", "ename": [ "Sato", "Takenaka", "Seno" ] }
{ "_id": "D02", "ename": [ "Washington", "Lincoln" ] }
{ "_id": "D91", "ename": [ "Suzuki", "Tanaka", "Matsuda" ] }
```

303

\$projectステージ

{ \$project: { 出力フィールドの指定 } }

出力のフィールドを指定する(findの出力フィールド指定と同じ)

```
db.emp.aggregate([
  { $match: { ename: "Smith" },
    { $project: { _id: 0, ename: 1, salary: 1 } }
])
```

結果

```
{ "ename": "Smith", "salary": 100 }
```

304

非正規形データへの対応

- 直積
 - 入れ子構造のオブジェクトにより表現可能
 - (例) {name: "基礎工学部",
 address: {street: "待兼山町1-3",
 city: "豊中市", zip: "560-8531"}}
- 巾集合
 - 配列により表現可能
 - (例) {name: ["大阪太郎", "大阪春子", "大阪小太郎"] }

305

(参考)正規化の例(直積)

(normalization of Cartesian product)

住所録(正規化前)

| 氏名 | 住所 |
|------|----------------------|
| 大阪太郎 | 560-8531大阪府豊中市待兼山町 |
| 神戸次郎 | 657-8501兵庫県神戸市灘区六甲台町 |

↓

住所録(正規化後)

| 氏名 | 郵便番号 | 都道府県 | 市区 | 町村 |
|------|----------|------|-------|------|
| 大阪太郎 | 560-8531 | 大阪府 | 豊中市 | 待兼山町 |
| 神戸次郎 | 657-8501 | 兵庫県 | 神戸市灘区 | 六甲台町 |

dom(住所)=dom(郵便番号) × dom(都道府県) × dom(市区) × dom(町村)

306

直積データの例

データベースへの挿入(直積データをオブジェクトの中に埋め込む)

```
db.school.insertOne( {name: "Engineering",
  address: {street: "2-1 Yamadaoka",
    city: "Suita", zip: "565-0871"} } )
db.school.insertOne( {name: "Science",
  address: {street: "1-1 Machikaneyama",
    city: "Toyonaka", zip: "560-0043"} } )
db.school.insertOne( {name: "Engineering Science",
  address: {street: "1-3 Machikaneyama",
    city: "Toyonaka", zip: "560-8531"} } )
```

307

直積データの問合せ例

問合せ(直積データ(オブジェクト)の中のフィールドを問合せで指定できる)

```
db.school.find({"address.city": "Toyonaka"}, {"name": 1})
```

結果

```
{ "_id": ObjectId("587db012c2da811995c28b95"),
  "name": "Science" }
{ "_id": ObjectId("587db021c2da811995c28b96"),
  "name": "Engineering Science" }
```

308

(参考)正規化の例(巾集合) (normalization of power set)

世帯(正規化前)

| 世帯 | 家族 |
|-----|-------------------|
| 世帯1 | {大阪太郎、大阪春子、大阪一太郎} |
| 世帯2 | {神戸次郎、神戸夏子、神戸小次郎} |

dom(家族) = 2^{dom(構成員)}

巾集合(power set)とは、ある集合のすべての部分集合からなる集合
例 ドメインD={1,2,3}とすると、Dの巾集合
2^D={ {}, {1}, {2}, {3}, {1,2}, {2,3}, {1,3}, {1,2,3} }となる

世帯(正規化後)

| 世帯 | 構成員 |
|-----|-------|
| 世帯1 | 大阪太郎 |
| 世帯1 | 大阪春子 |
| 世帯1 | 大阪一太郎 |
| 世帯2 | 神戸次郎 |
| 世帯2 | 神戸夏子 |
| 世帯2 | 神戸小次郎 |

309

巾集合データの例

データベースへの挿入(配列を使って集合を表現できる)

```
db.registration.insertOne( {subject: "OS",
  lecturer: ["Murata", "Matsuda"]} )
db.registration.insertOne( {subject: "Database",
  lecturer: "Matsuda"} )
```

310

巾集合データの問合せ例

問合せ

```
db.registration.find({lecturer: "Matsuda"},
  {subject: 1})
```

結果

```
{ "_id": ObjectId("587db73fc2da811995c28b97"),
  "subject": "OS" }
{ "_id": ObjectId("587db750c2da811995c28b98"),
  "subject": "Database" }
```

311

MongoDBでのMapReduce処理(1)

- データベースのデータ

```
db.Orders.insertOne({"Name": "notebook",
  "Price": 200, "Category": "material"}, ...)
db.Orders.insertOne({"Name": "bread",
  "Price": 100, "Category": "food"}, ...)
```

- 結果

```
{ "_id": "food",
  "value": {"Category": "food", "Count": 2, "Amount": 320} }
{ "_id": "material",
  "value": {"Category": "material", "Count": 3, "Amount": 300} }
```

出典 開発者が知っておくべき、ドキュメント・データベースの基礎
<http://www.atmarkit.co.jp/ait/articles/1211/09/news056.html>

312

(参考) MapReduceの実装

出典 "Turn" Introduction to HBase Technology,
URL: <http://www.programering.com/a/MjNwQjNwATQ.html>

313

MongoDBでのMapReduce処理(2)

mapの関数とreduceの関数をユーザが定義して処理する

```
function mapf() {
  emit(this.Category,
  {Category:this.Category, Count:1, Amount:this.Price})
}
function reducef(key, values) {
  var result = {Category:key, Count:0, Amount:0};
  values.forEach(function(v) {
    result.Count += v.Count;
    result.Amount += v.Amount; });
  return result; }
```

314

MongoDBでのMapReduce処理(3)

- 実行
`db.Orders.mapReduce(mapf, reducef, {out: {replace:"testcol"}})`
- 実際に分散サーバ環境で実行するには、次に説明するshardingを利用する

315

MongoDBの分散環境での実行 (sharding)

クライアントからの処理要求をルータサーバで振り分けて複数のデータベースサーバに割り付ける
 出典 MongoDB Documentation <https://docs.mongodb.com/>

316

Neo4j

- グラフ型のNoSQLデータベース
- データベースのデータを、**節点(node)**と**辺 (relationships)**からなる**グラフ**で表現、節点と辺には**属性(property)**を付与できる
- CypherというSQLに似たデータ操作言語を備える
- 問合せでは**グラフを探索**(複数の節点や辺にまたがる結合問合せを表現可能)
- ACID特性を持つトランザクション(ただしC(整合性)は複数サーバで使うときは**結果整合性**)
- 大規模なグラフデータを高速に検索可能**(10億ノードのグラフを数分で検索、MapReduceではなく独自の複数サーバ分散技術(causal clustering)を利用)

317

リレーショナルデータベースとの比較

emp

| empno (P) | ename | deptno (R) | salary | roomno (R) |
|-----------|--------|------------|--------|------------|
| E01 | Smith | D01 | 100 | R01 |
| E02 | Morgan | D01 | 70 | R02 |

dept

| deptno (P) | dname | manager (R) |
|------------|---------|-------------|
| D01 | Account | E01 |

room

| roomno (P) | rname | deptno (R) |
|------------|----------------|------------|
| R01 | Account Room 1 | D01 |
| R02 | Account Room 2 | D01 |

(注) (P)は主キー、(R)は外部キーを表す

Neo4jでのデータベース 318

(注) タプルを節点(node)、外部キーを辺(relationship)で表現する

Neo4jでのデータ登録例 319

- CREATE文によりデータを登録する
- 節点データの登録
- CREATE (変数:ラベル {属性リスト})
- 辺データの登録
- CREATE (変数1)-[:ラベル]->(変数2)

(例) (注) E01, E02, D01, R01, R02はIDではなく登録時にのみ有効な一時的な変数

```
CREATE (E01:emp {ename: 'Smith', salary: 100}),
      (D01:dept {dname: 'Account'}),
      (R01:room {rname: 'Account Room 1'}),
      (E01)-[:deptno]->(D01), (E01)-[:roomno]->(R01),
      (D01)-[:manager]->(E01), (R01)-[:deptno]->(D01)
CREATE (E02:emp {ename: 'Morgan', salary: 70}),
      (R02:room {rname: 'Account Room 2'}),
      (E02)-[:deptno]->(D01), (E02)-[:roomno]->(R02),
      (R02)-[:deptno]->(D01)
```

Neo4jでの問合せ 320

- MATCH文により問合せを記述する
- 節点に対する問合せ
- MATCH (変数:ラベル) WHERE 検索条件 RETURN 検索対象
- 辺に対する問合せ (SQLの結合問合せに相当する)
- MATCH (変数1:ラベル1)-[:ラベル3]->(変数2:ラベル2) WHERE 検索条件 RETURN 検索対象

(例)

```
MATCH (e:emp) WHERE e.salary >= 100
RETURN e.ename
MATCH (e:emp)-[:deptno]->(d:dept)
WHERE e.ename = 'Smith' RETURN d.dname
MATCH (e:emp)-[:deptno]->(d:dept)
WHERE d.dname = 'Account' RETURN e.ename
```

(注) 属性の条件を属性リストを使って書くことも可能

```
MATCH (e:emp {ename: 'Smith'})-[:deptno]->(d:dept)
RETURN d.dname
```

グラフ探索の例(1) 321

- 経路を求める

```
MATCH p = ((e1:emp)-[*]->(e2:emp))
WHERE e1.ename = 'Morgan' AND e2.ename = 'Smith'
RETURN p
```

(別の記法: MATCH文の属性条件を属性リストで表記)

```
MATCH p = ((e1:emp {ename: 'Morgan'})-[*]->(e2:emp {ename: 'Smith'})) RETURN p
```

(結果) 複数の経路が表示される

```
[{"salary": 70, "ename": "Morgan"}, {"dname": "Account"}, {"rname": "Account Room 2"}, {"rname": "Account Room 2"}, {"dname": "Account"}, {"dname": "Account"}, {"salary": 100, "ename": "Smith"}]
[{"salary": 70, "ename": "Morgan"}, {"dname": "Account"}, {"dname": "Account"}, {"salary": 100, "ename": "Smith"}]
```

グラフ探索の例(2) 322

- 最短経路を求める

```
MATCH p = shortestPath((e1:emp)-[*]->(e2:emp))
WHERE e1.ename = 'Morgan' AND e2.ename = 'Smith'
RETURN p
```

(別の記法: MATCH文の属性条件を属性リストで表記)

```
MATCH p = shortestPath((e1:emp {ename: 'Morgan'})-[*]->(e2:emp {ename: 'Smith'})) RETURN p
```

(結果) 最短経路のみが表示される

```
[{"salary": 70, "ename": "Morgan"}, {"dname": "Account"}, {"dname": "Account"}, {"salary": 100, "ename": "Smith"}]
```

Neo4jの複数サーバでの利用例 323

複数のクライアント(App servers)からのアクセスは、更新を単一のサーバ(core)に、問合せをデータベースの複製サーバ(Read replica)に分けることで性能を向上。複数の更新が独立に生じると、クライアントごとに読む複製の値が一時的に異なることが起こり得るが、十分長い時間に更新がなければ同じ値になる(結果整合性)