CSCI235 Final Cheat sheet

1NF:

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.

2NF

- In the second normal form, all non-key attributes are fully functional dependent on the primary key

3NF

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- X is a super key. X->Y
- Y is a prime attribute (each element of Y is part of some candidate key).
- If A->B and B->C are two FDs then A->C is called transitive dependency.

BCNF:

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

4NF:

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

5NF:

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).
- In the fifth normal form the relation must be decomposed in as many sub-relations as possible so as to avoid any kind of redundancy and there must be no extra tuples generated when the sub-relations are combined together by using natural join.

3NF Synthesis Algorithm

1. **Make right hand side atomic (1NF)**
2. **Elimination of Extraneous Attributes**
    1. Reduction rule 1: XY --> Z and X --> Z then Y is extra on left side.
    2. Reduction rule 2: XY --> Z and X --> Y then Y is extra we need only X --> Z.
    3. For example: DEK --> JF and K → E then E is extra we need only DK → JF
3. **Search for Non redundant cover**

    A --> E, E --> G, A --> G so remove A --> G
4. **Partition of nonredundant cover**

    Groups with same left side in one group

    non-fully functional dependencies should be in separate groups.
5. **Merger of equivalent keys**
6. **Definition of Tables**

XPath

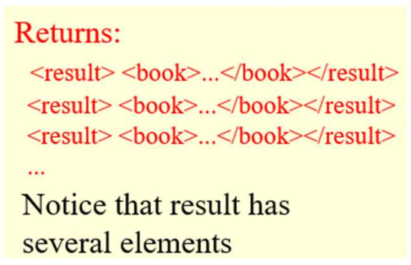
Select title nodes for the author “Erik T. Ray “ with price>35

```
//book[price >35 and author = "Erik T. Ray"]/title
```

Note: /salesdata/year[2]/theyear   picks only 2nd year element.

XQuery

“for” returns

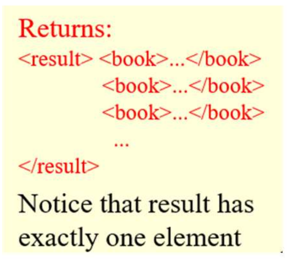


Returns:

```
<result> <book>...</book></result>
<result> <book>...</book></result>
<result> <book>...</book></result>
...
```

Notice that result has several elements

“let” returns

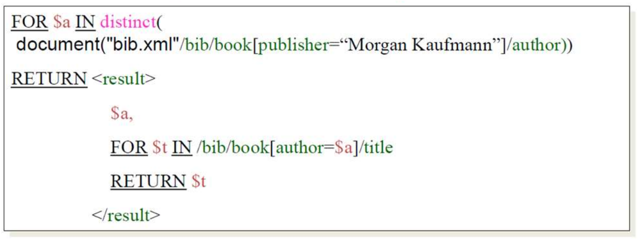


Returns:

```
<result> <book>...</book>
         <book>...</book>
         <book>...</book>
         ...
</result>
```

Notice that result has exactly one element

For each author of a book by Morgan Kaufmann, list all books she published:



```
FOR $a IN distinct(
 document("bib.xml"/bib/book[publisher="Morgan Kaufmann"]/author))
RETURN <result>
           $a,
           FOR $t IN /bib/book[author=$a]/title
           RETURN $t
       </result>
```

distinct = a function that eliminates duplicates (after converting inputs to atomic values)

Q. For parts manufactured by ASUS return their cost and for parts manufactured by Creative Labs return the model name.

```
for $x in //PART
  return
 if ($x/MANUFACTURER = "ASUS") then
   <MANUFACTURER> {$x/MANUFACTURER}
   <price> {$x/COST}</price></MANUFACTURER>
else if ($x/MANUFACTURER = "Creative Labs")
 then <MANUFACTURER> {$x/MANUFACTURER}
   <price> {$x/MODEL}</price></MANUFACTURER>
   else ()
```

Q. Find all items.

If the file is not loaded

```
$x = doc(“PARTS.XML”)/ITEM
RETURN $x
```

If the data is in the database

```
for $x in //PART
return $x/ITEM
```

```
<ITEM>Motherboard</ITEM>
<ITEM>Video Card</ITEM>
<ITEM>Sound Card</ITEM>
<ITEM>19 inch Monitor</ITEM>
```

Q. Find all items that cost more than 150.

```
for $x in //PART
where $x/COST > 150
return $x/ITEM
```

```
<ITEM>Video Card</ITEM>
<ITEM>19 inch Monitor</ITEM>
```

Q. Find the cost of all items from ASUS.

```
for $x in //PART
where $x/MANUFACTURER = "ASUS"
return <M> $x/MANUFACTURER <cost> $x/COST</cost> </M>
```

Indexing

Clustered Index: A clustered index is a special type of index that **reorders the way records in the table are physically stored**. Therefore, table can have only one clustered index.

B-Tree Index: B-tree index is preferable for range queries.

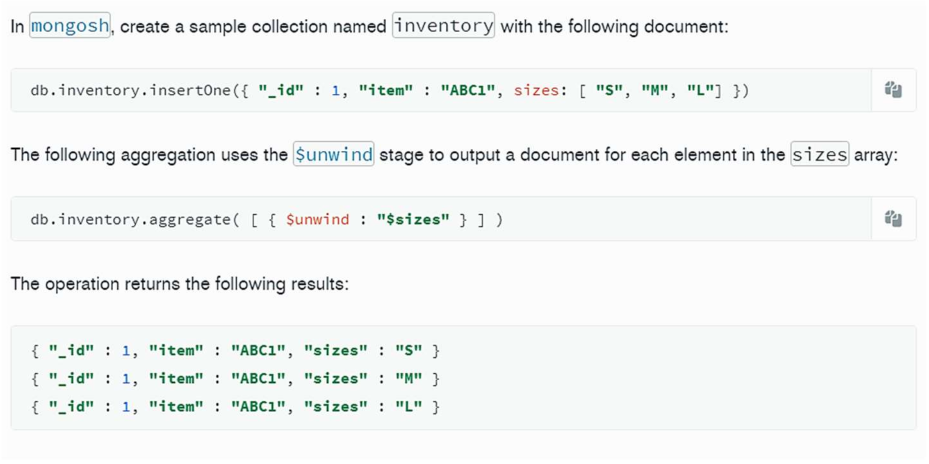
Hash index: Hash index is preferable for equality queries. A hash index is an index type that is most commonly used in data management. It is typically created on a column that contains unique values, such as a primary key or email address.

Mongo dB

The $push operator appends a specified value to an array.

If you use $addToSet on a field that is **not** an array, the operation will fail.

$unwind Deconstructs an array field from the input documents to output a document for *each* element. Each output document is the input document with the value of the array field replaced by the element.


In mongosh, create a sample collection named inventory with the following document:

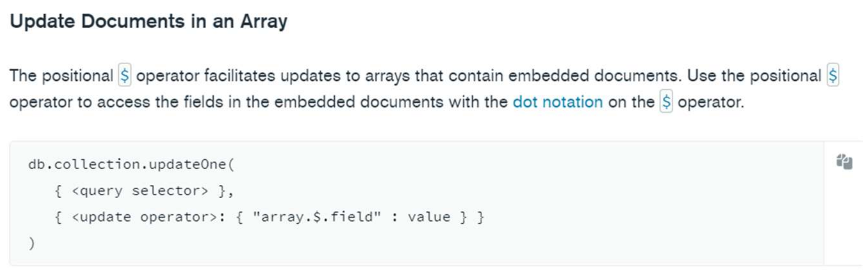
```
db.inventory.insertOne({ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] })
```

The following aggregation uses the $unwind stage to output a document for each element in the sizes array:

```
db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
```

The operation returns the following results:

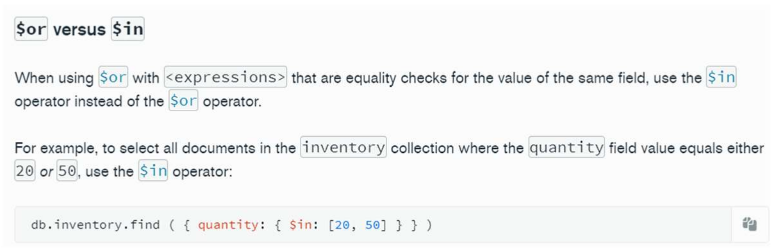
```
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```


Update Documents in an Array

The positional $ operator facilitates updates to arrays that contain embedded documents. Use the positional $ operator to access the fields in the embedded documents with the dot notation on the $ operator.

```
db.collection.updateOne(
   { <query selector> },
   { <update operator>: { "array.$.field" : value } }
)
```

Note: $slice: limits the number of elements in an array.


$or versus $in

When using $or with <expressions> that are equality checks for the value of the same field, use the $in operator instead of the $or operator.

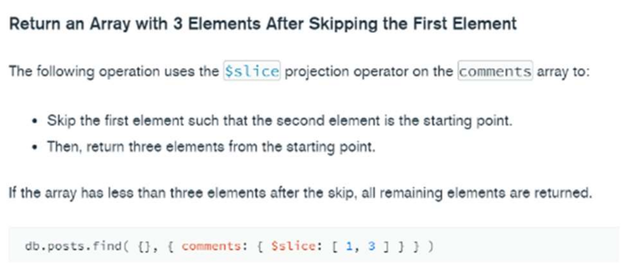
For example, to select all documents in the inventory collection where the quantity field value equals either 20 or 50, use the $in operator:

```
db.inventory.find ( { quantity: { $in: [20, 50] } } )
```

The limit() method is used to restrict the number of documents displayed to the user.

Q10. Find cars with the speed of bigger than 40 and return first two of them sorted based on name.

db. collection.find({speed: {$gt:40} }).sort({name:1}).limit(2)


Return an Array with 3 Elements After Skipping the First Element

The following operation uses the $slice projection operator on the comments array to:

- Skip the first element such that the second element is the starting point.
- Then, return three elements from the starting point.

If the array has less than three elements after the skip, all remaining elements are returned.

```
db.posts.find( {}, { comments: { $slice: [ 1, 3 ] } } )
```

Note: The $slice projection operator specifies the number of elements in an array to return in the query result.

Q. write a MongoDB.find() query with elemmatch to find plants that the price of their nursery is bigger than 2.5 AUD or 2 USD. Return their common name and number of their nurseries (count) but no ID.

```
  db.collection.find({ "plant.availability": {
   "$elemMatch": {
     "$or": [
       {
         "nursery.price": {
           "$gt": 2},
         "nursery.currency": "AUD"},
       {
         "nursery.price": {
           "$gt": 2.5 },
         "nursery.currency": "USD"  }
     ]
   } }
},
{ "_id": 0, "plant.common name": 1, "number of their nurseries ": { "$size": "$plant.availability"}
})
```

Q. write a MongoDB.aggregate() query to find the plants that their nursery price is in AUD. Return the common name and nursery name of the plant.

```
          db.collection.aggregate([
            {
              "$unwind": "$plant.availability"
            },
            {
              "$match": {
                "plant.availability.nursery.currency": "AUD"
              }
            },
            {
              "$project": {
                "plant.availability.nursery.name": 1,
                "plant.common name": 1
              }
            }
  ])
```

Question. Show movie name, release date, director and the country that generated revenues between 100000000 to 250000000

```
db.movie.aggregate([{
  $unwind:"$Revenue"},
  {
  $match:{"Revenue.Sales":{"$gt":100000000,"$lt":250000000} }
  },
{$project: {"Name":1, "Date":1, "Director":1, "Revenue.Country":1}
}
])
```

Question: change the name of actor of movie Dont Breath to Russle Nesson and add following awards to it BAFTA 1980 for Best Director and Best Audio Visual

```
db.movie.updateMany(
    {
        "Name":"Dont Breath"
    },
    {$set: {"Actor": "Russle Nesson"},
    $push: {"Awards":{"Name":"BAFTA","Year": 1980,
    "Categories":["Best Director","Best Audio Visual"]}}
    }
)
```

Question: change UAE to India for all movies

```
db.movie.updateMany(
   {"Revenue.Country": "UAE" },
   { $set: {"Revenue.$.Country" : "India" } }
)
```

Question: show movie name, date, actor, and director for all movies directed by John Luis or Adam Ibra but not acted by Richard Robertson

```
/*Task 7*/
db.movie.find({
  "Director" : {$in : ["Adam Ibra","John Luis"]},
   "Actor" : {$ne : "Richard Robertson" }},
  {_id:0, Name:1, Date:1, Actor:1, Director:1}
  )
```

Although you can write this query using the `$or` operator, use the `$in` operator rather than the `$or` operator when performing equality checks on the same field.

```
/*T3.5 Find teachers teaching more than 2 courses*/

db.lab7CSCI235.find({"Courses":{$size:3}},{"Name.$":1,"_id":0})
/*Either*/
db.lab7CSCI235.find({"Courses.2":{"$exists":true}},
{"Name":1, "_id":0}
)
```

```
/*TASK 4 change UAE to India for all movies*/
db.movie.updateMany(
   {"Revenue.Country": "UAE" },
   { $set: {"Revenue.$.Country" : "India" } }
)
```

Question: find all female employees who are responsible for Daily Support or Mobile Application

```
db.lab7CSCI235.find({Gender:"F",$or:[{"Job List":{$elemMatch:{$eq:"Daily Support"}}}
,{"Job List":{$elemMatch:{$eq:"Mobile Application"}}}]},{"_id":0})
```

```
db.lab7CSCI235.find({"Gender":"F",
$or:[{"Job List":"Daily Support"},{"Job List":"Mobile Application"}]})
```

Question: find all female employees who are responsible for Daily Support and Mobile Application

```
/*Without $and*/
db.lab7CSCI235.find({Gender:"F","Job List":{$elemMatch:
{$eq:"Daily Support",$eq:"Mobile Application"}}})
```

```
/*with $and*/
db.lab7CSCI235.find({Gender:"F",$and:[{"Job List":{$elemMatch:
{$eq:"Daily Support",$eq:"Mobile Application"}}}]})
```

Distributed Database

On server projects.com we have the following two tables:

Project (**projNo**, projName, ProjMgrId, budget, startDate, expectedDurationWeeks)

Assign(**projNo, empId**, hoursAssigned, rating)

On server workers.com we have the following table:

Worker(**empId**, lastName, firstName, departmentName, birthDate, hireDate, salary)

Now, assume you are running the following command from a third database.

SELECT p.projName, W.lastName, A.hoursAssigned

FROM Project P, Assign A, Worker W

WHERE P.projID = A.projID AND A.empId = W.empId

Asumming your username is John and password is P@ssW0rd and

- The table Project is on the database projects.com with the port 1521 and the database service name is Projects_production.
- The table Worker is on the database server workers.com with the port number 1522 and the database service name is Workers_all.

Write appropriate commands to create a Fixed User database link. You need to create two entries in tnsnames.ora file, two CREATE DATABASE LINKS and two synonym.

1. **create database link link_name connect to John identified by P@ssW0rd using 'Projects_production';**
2. **create database link dblink connect to John identified by P@ssW0rd using 'Workers_all';**

Database link for the 1st one

```
CREATE DATABASE LINK dblink1 CONNECT TO John IDENTIFIED BY P@ssW0rd
USING '(DESCRIPTION= (ADDRESS=(PROTOCOL=TCP)(HOST=projects.com)
    (PORT=1521))
    (CONNECT_DATA=(SERVICE_NAME=Projects_production)) )' ;
```

For Public database link, example:

```
CREATE PUBLIC DATABASE LINK dblink
  CONNECT TO user1 IDENTIFIED BY password2
  USING 'remote_database';
```

For Update a table on the remote database:

```
UPDATE employees@dblink
  SET salary=salary*1.1
 WHERE last_name = 'Baer';
```

Partitioning Tables in Oracle

Sailors(sid: **integer,** *sname:* **varchar,** *rating:* **integer,** *age:* **integer)**

a. Write appropriate commands to create a range partitioned table for the Sailors table. The attribute sid ranges from 1001 to 2000. Assume you want to create 4 groups of 250 sailors each.

```
(CREATE TABLE Sailors
(sid  NUMBER(5),
sname VARCHAR2(30),
rating  NUMBER(2),
age  NUMBER(2))
PARTITION BY RANGE(sid)
PARTITION sid_g1 VALUES LESS THAN(1250),
PARTITION sid_g2 VALUES LESS THAN(1500),
PARTITION sid_g3 VALUES LESS THAN(1750),
PARTITION sid_g4  VALUES LESS THAN(2000)    );
```

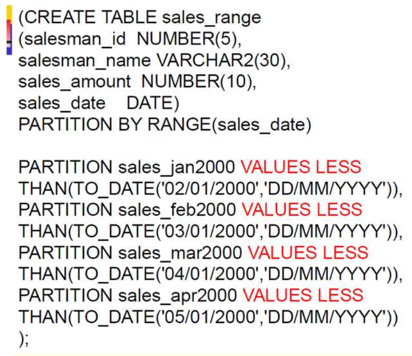
b. Write appropriate commands to create a global Hash Partitioned Index for this table as well.

```
CREATE INDEX sid_idx ON Sailors (sid)
GLOBAL PARTITION BY HASH(sid)
PARTITIONS 4 STORE IN (data1, data2, data3, data4) ;
```

- **Global Partition Indexes:**

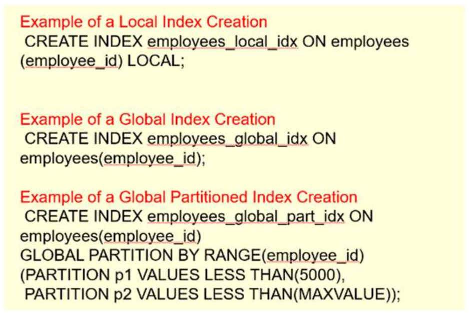
- Global partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method, commonly used for OLTP environments and offer efficient access to any individual record.
- You cannot add a partition to a global index because the highest partition always has a partition bound of MAXVALUE. You cannot drop the highest partition in a global index.
- If a global index partition contains data, dropping the partition causes the next highest partition to be marked unusable.
- If you wish to add a new highest partition, use the ALTER INDEX SPLIT PARTITION statement. If a global index partition is empty, you can explicitly drop it by issuing the ALTER INDEX DROP PARTITION statement.
- Global nonpartitioned indexes behave just like a nonpartitioned index. They are commonly used in OLTP environments and offer efficient access to any individual record.

Range Partitioning



```
(CREATE TABLE sales_range
(salesman_id  NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount  NUMBER(10),
sales_date    DATE)
PARTITION BY RANGE(sales_date)

PARTITION sales_jan2000 VALUES LESS
THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
PARTITION sales_feb2000 VALUES LESS
THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LESS
THAN(TO_DATE('04/01/2000','DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS
THAN(TO_DATE('05/01/2000','DD/MM/YYYY'))
);
```



Example of a Local Index Creation

```
CREATE INDEX employees_local_idx ON employees
(employee_id) LOCAL;
```

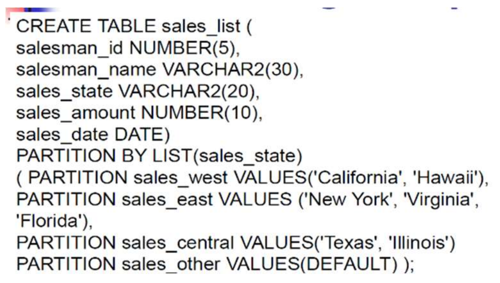
Example of a Global Index Creation

```
CREATE INDEX employees_global_idx ON
employees(employee_id);
```

Example of a Global Partitioned Index Creation

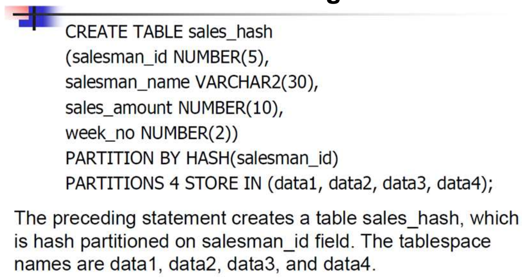
```
CREATE INDEX employees_global_part_idx ON
employees(employee_id)
GLOBAL PARTITION BY RANGE(employee_id)
(PARTITION p1 VALUES LESS THAN(5000),
 PARTITION p2 VALUES LESS THAN(MAXVALUE));
```

List Partitioning



```
CREATE TABLE sales_list (
salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_state VARCHAR2(20),
sales_amount NUMBER(10),
sales_date DATE)
PARTITION BY LIST(sales_state)
( PARTITION sales_west VALUES('California', 'Hawaii'),
PARTITION sales_east VALUES ('New York', 'Virginia',
'Florida'),
PARTITION sales_central VALUES('Texas', 'Illinois')
PARTITION sales_other VALUES(DEFAULT) );
```

Hash Partitioning



```
CREATE TABLE sales_hash
(salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount NUMBER(10),
week_no NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4 STORE IN (data1, data2, data3, data4);
```

The preceding statement creates a table sales_hash, which is hash partitioned on salesman_id field. The tablespace names are data1, data2, data3, and data4.

Differences between READ COMMITTED AND SERIALIZABLE

READ COMMITTED

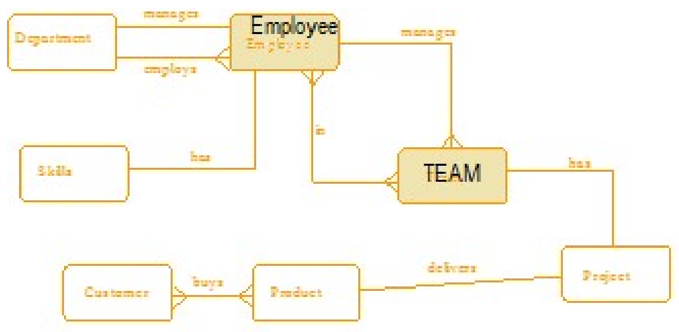
- Each query executes with respect to its own materialized view time thereby permitting nonrepeatable reads and phantoms for multiple executions of a query
- Recommended when few transactions are likely to conflict
- Default isolation level

SERIALIZABLE

- If a Transaction T is running at tis level tries to update or delete data modified by a transaction that commits after the serializable Transaction T began the system aborts the Transaction T.
- If a serializable transaction fails, then it is possible to:
  - Commit the work executed to that point
  - Execute additional (but different) statements
  - Rollback the entire transaction

BSON Object Modelling

Q. Design one or more BSON documents to represent the following Hierarchical relationship among Department-Employee-Team many to many relationship.

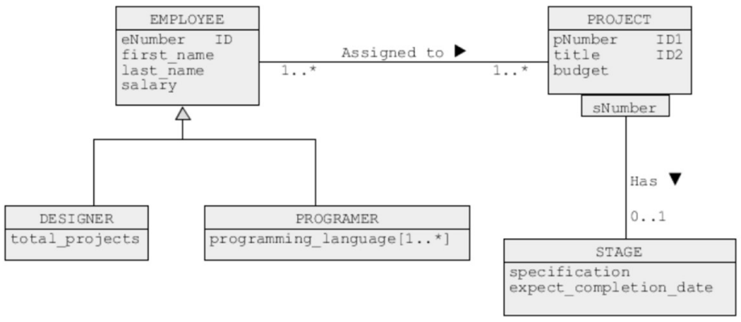


Department has many employees and each employee manages many teams.
Employee has a name and ID. Team also has name and ID.
Employee 101, Smith is manages Team 1 Database and Team 2 Data Mining.
Employee 102, Jones is manages Team 3 Project Management.
Employee 101 belongs to Department 1 Accounting. Employee 102 belongs to Department 2.

```
{ "Department": { "ID": "1",
          "Name": "Accounting",
          "Employee": { "ID":  "101",
                  "Name": "Smith",
                  "Manages": [{ "Team": { "ID":  "2",
                                    "Name": "Data Mining"}
                                    },
                                      { "Team": { "ID":  "2",
                                    "Name": "Data Mining"}
                                }]
                             }
                }
}

{ "Department": { "ID": "2",
          "Name": "Marketing",
          "Employee": { "ID":  "102",
                  "Name": "Jones",
                  "Manages": [{ "Team": { "ID":  "3",
                                    "Name": "Project Management"}
                                    }]
                             }
                }
}
```

Consider the following object model:



Task2: How would you store the data in the above object model in a NOSQL document-based database using XML or BSON or JSON?

```
{Employee:{_id:$eNumber,
          "firstName":
          "last_name":
          "salary":
          "Designer":{"total_projects"}
          "programmer":{"programming_language":["Java", "Python"]}
}
}
{project:{_id:$pNumber,
            "title":
            "budget":
           Stage:{"sNumber":
                  "specification":
                  "expect_completion_date":}
                  }}
{Assigned to:{_
id: {$concat:{$eNumber, pNumber}}
}
```

<u>In XML</u>

```
<Employee>
          <eid> 101 </eid>
          <firstName> xxx </firstname>
         <lastName> yyyy </lastName>
         <salary> 20000 </salary>
         <Designer>
                  <total Projects> 5 </total Projects>
         </Designer>
</Employee>
<Project>
         <pNumber>1 </pNumber>
          <title> project 1 </title>
         <budget> 20000000 </budget>
         <Stage>
                  <StageNo> s1 </StageNo>
                 <specification> xxxxxxxxxx </specification>
                 <expect_completion_date> 1/2/2020 </expect_completion_date>
         </Stage>
</Project>
<Employee_project>
         <eid> 101 </eid>
         <pNumber>1 </pNumber>
</Employee_project>
```

Note: It is possible for two items to have the same partition key value, but those two items must have different sort key values.

Q. How would you store the data in the above object model in NOSQL key-value database?

Project StageTable

Project items: {partition key: pNumber, Sort Key: Null} Value: {"employees":[] }

Stage items: {partition key: pNumber, sort key: sNumber}

Employee Table

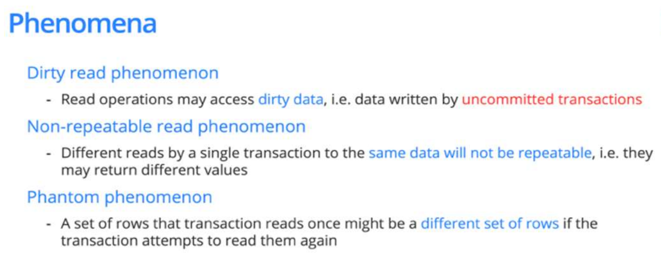
Employee item: {P key: eNumber, Sort key: NULL} Value{"projects":[] }

Designer item {P key: eNumber, Sort key: "D"}

Programmer item {P Key: eNumber, Sort Key: "P"}

Assigned to item: {P key: eNumber, Sort key: pNumber}

Transaction



Phenomena

Dirty read phenomenon

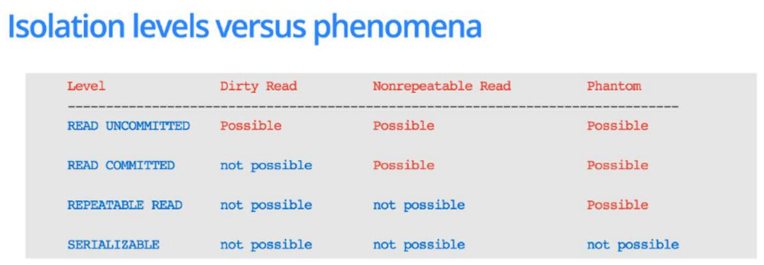
- Read operations may access dirty data, i.e. data written by uncommitted transactions

Non-repeatable read phenomenon

- Different reads by a single transaction to the same data will not be repeatable, i.e. they may return different values

Phantom phenomenon

- A set of rows that transaction reads once might be a different set of rows if the transaction attempts to read them again

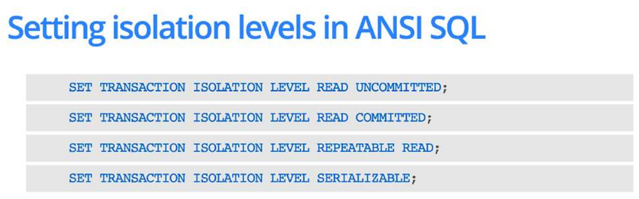


Isolation levels versus phenomena

Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	not possible	Possible	Possible
REPEATABLE READ	not possible	not possible	Possible
SERIALIZABLE	not possible	not possible	not possible

The REPEATABLE READ transaction will still see the same data, while the READ COMMITTED transaction will see the changed row count.

Note: REPEATABLE READ and SERIALIZABLE will not see any new changes in the table.



Setting isolation levels in ANSI SQL

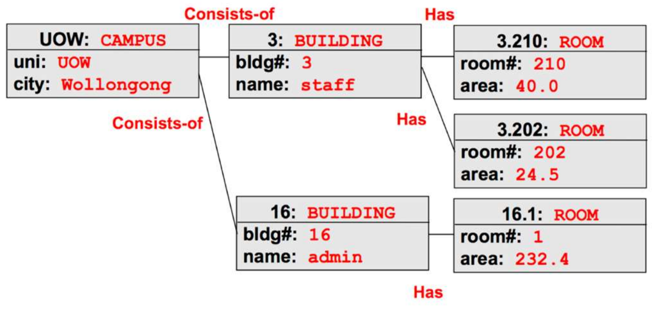
```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Note: if it's "for update" in the select stxxatement, then it will be REPEATABLE READ.

Convert the following XML document to BSON document.

In your implementation, make sure the Price tag has two objects 'currency" and "value". Make sure discount has an array of objects. Each object has a "type", and a "value".

```
<prices>
  <priceList effDate="2015-11-15">
    <prod num="557">
      <price currency="USD">29.99</price>
      <discount type="CLR">10.00</discount>
    </prod>
    <prod num="563">
      <price currency="USD">69.99</price>
    </prod>
    <prod num="443">
      <price currency="USD">39.99</price>
      <discount type="CLR">3.99</discount>
    </prod>
  </priceList>
</prices>
{
 "Price":  { "priceList": { "effDate":"2015-11-15",
                    "prods": [  {"prod": { "prodnum":"557",
                                   "price": { "currency":"USD",   "value":"29.99" },
                                   "discount": [ {"type":"clr" ," value":"10.00"} ]
                                  }
                          },
                          {"prod": { "prodnum":"563",
                                   "price": {"currency":"USD","value":"69.99"}
                                  }
                           },
                          {"prod": { "prodnum":"443",
                                   "price": {"currency":"USD", "value":"39.99"},
                                   "discount":[ {"type":"clr", " value":"3.99"}]
                                  }
                          }
                        ]
                    }
         }
}
```



Simplified

```
  {"uni":"UOW",
   "city":"Wollongong",
   "Consists of": [ {"bldg#":3,
                     "name":"staff",
                     "Has":[ {"room#":210,
                              "area":40.0},
                             {"room#":202,
                              "area":24.5}
                           ]
                    },
                    {"bldg#":16,
                     "name":"admin",
                     "Has":[ {"room#":1,
                              "area":232.4}
                           ]
                    }
                  ]
  }
```

Sharding

Basics

Sharding is the process of partitioning a large dataset into smaller and more manageable pieces

A shared nothing architecture is a distributed computing architecture in which each computing node does not share data with other the nodes

In database systems it is called as sharding (shared nothing)

What to do when:

- large amount of data and greater read/write throughput demands make commodity database servers not sufficient ?
- the database servers are not be able to address enough RAM, or they might not have enough CPU cores, to process the workload efficiently ?
- due to large amount of data it is not practical to store and to manage backups on one disk or RAID storage (Redundant Array of Inexpensive Disks) ?

Why sharding ?

There are two main reasons to shard:

- storage distribution
- load distribution

If monitoring of storage capacity shows that at certain moment the database applications require more storage than it is available and adding more strage is impossible then sharding is the best option

Mongodb monitoring means running db.stats() and db.collection.stats() in the mongo shell to get the statistics about the storage usage of the current database and collection within it

Load means CPU and RAM utilization, I/O bandwidth, network transmission used by the requests from the clients => response time

If at certain moment response time does not match the client's expectations then it triggers a decision to shard

Therefore, a decision to shard an existing system should be based on analyses of network usage, disk usage, CPU usage, and ratio the amount of data actively being used to available RAM

Sharding in MongoDB is **range-based.** It means that each chunk represents a range of **shard** keys. To determine what **Chunk** a document belongs to , MongoDB extracts the values for a **shard key** and then finds a **chunk** whose **key range** contains the given shard.

Enable sharding of a database

```
sh.enableSharding("test")                                    enableSharding()
db.getSiblingDB("config").databases.find()
```

Shard a collection

```
sh.shardCollection("test.testcol", {"shard-key": 1})         shardCollection()
```

Types of NoSQL Databases:

Key-Value,
Document databases
Column family stores
Graph Databases

Replication

Replication provides safety mechanisms protecting database system from environment failures like:

- a network connection between the application and the database is lost,
- there is a loss of power,
- persistent storage device (HDD, SSD) fails

In addition to protecting against external failures, replication is important for system durability

When running without backup/journaling the original values of corrupted data cannot be easily restored

Replication can always guarantee a clean copy of the data files if a single node shuts down due to a hardware fault

Replication also facilitates redundancy, failover, maintenance, and load balancing

Replication is designed primarily for redundancy

Commit and rollback

Writes are not considered as committed until they are replicated to a majority of nodes

Operations on a single document are always atomic and operations that involve multiple documents are not atomic

Consider the following scenario:

- a series of writes to the primary node did not get replicated to the secondary node,
- primary node goes offline and the secondary is promoted to primary and new writes go to primary
- old primary comes back online as secondary and tries to replicate from the new primary
- old primary has a series of writes that don't exist in the new primary node oplog
- It triggers a rollback

Rollback reverses all writes that were never replicated to a majority of nodes

How does replication work ?

Replica sets rely on two basic mechanisms: an oplog and a heartbeat

Oplog (operation log) is a space restricted collection that lives in a database called local on every replica node and records all changes to the data

Every time a client writes to the primary node, an entry with enough information to reproduce the write is automatically added to the primary node's oplog.

Once the write is replicated to a given secondary node then its oplog also stores a record of the write

When a given secondary node is ready to update itself, it performs the following actions:

- first, it looks at the timestamp of the latest entry in its own oplog
- next, it queries the primary node's oplog for all entries greater than that timestamp
- finally, it writes the data and adds each of those entries to its own oplog

Distribution Models:

- There are two styles of distributing data:
- **Sharding**: Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
- **Replication**: Replication copies data across multiple servers, so each bit of data can be found in multiple places. Replication comes in two forms,
  - **Master-slave replication** makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
  - **Peer-to-peer replication** allows writes to any node; the nodes coordinate to synchronize their copies of the data.
- Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single server creating a single point of failure. A system may use either or both techniques. Like Riak database shards the data and also replicates it based on the replication factor.

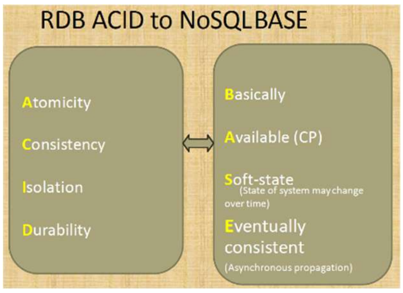
Weak consistency

It is possible tolerate long inconsistency window under a condition that read-your-write consistency is enforced

This leads to a concept of session consistency, which means that within a user session read-your-write consistency is enforced.

To provide session consistency it is possible to implement sticky session, i.e. a session that is attached to only one node in a cluster (it is also called as session affinity)

Another solution to provide session consistency is to use version timestamps where every interaction with a data item is performed on a data item with the highest timestamp



RDB ACID to NoSQLBASE

Atomicity	Basically
Consistency	Available (CP)
Isolation	Soft-state (State of system may change over time)
Durability	Eventually consistent (Asynchronous propagation)

- **Atomic**
  Either all database changes for an entire transaction are completed or none of the changes are completed.
- **Consistent**
  Database changes transform from one consistent database state to another.
- **Isolated**
  Transactions from concurrent applications do not interfere with each other. The updates from a transaction are not visible to other transactions that execute concurrently until the transaction commits.
- **Durable**
  Complete database operations are permanently written to the database

Weak consistency

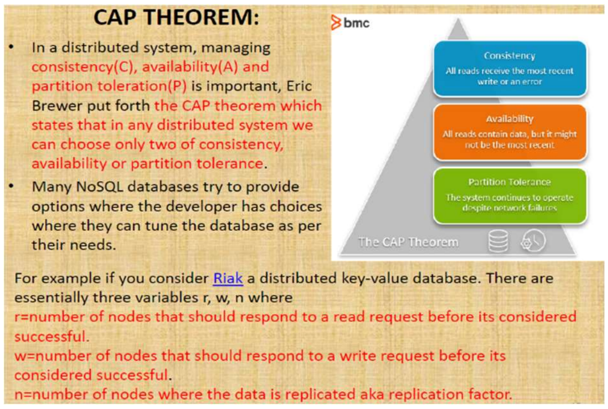
A typical read consistency principle where update is performed over two or more data items blocks access to all data items affected by the update, e.g. 2PL protocol

NoSQL database systems relax the transactional consistency to some extent

Data items can be left inconsistent over certain period of time called as inconsistency window

A concept of eventual consistency is used to enforce replication consistency over distributed and replicated data items

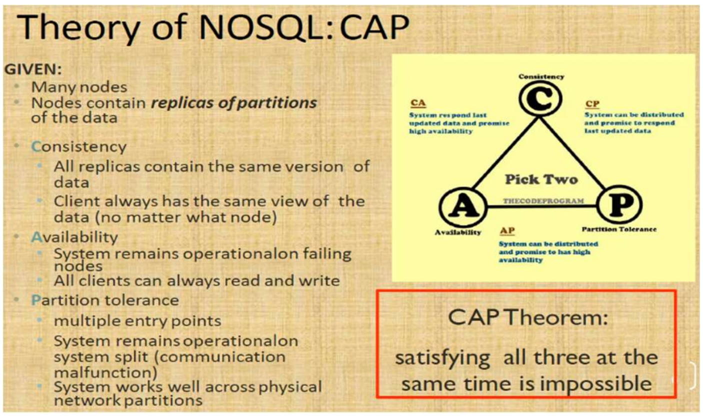
Eventual consistency means that the copies of data items can be inconsistent in inconsistency window and all copies will have the same value later on


bmc
Consistency
All reads receive the most recent write or an error
Availability
All reads contain data, but it might not be the most recent
Partition Tolerance
The system continues to operate despite network failures
The CAP Theorem

CAP THEOREM:

- In a distributed system, managing consistency(C), availability(A) and partition toleration(P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance.
- Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs.

For example if you consider Riak a distributed key-value database. There are essentially three variables r, w, n where
r=number of nodes that should respond to a read request before its considered successful.
w=number of nodes that should respond to a write request before its considered successful.
n=number of nodes where the data is replicated aka replication factor.

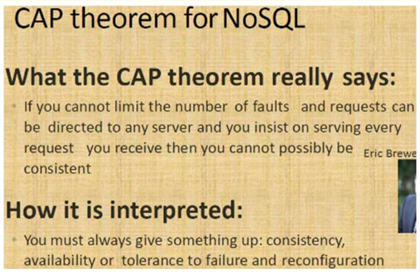

Consistency
C
CA
System respond last updated data and promise high availability
CP
System can be distributed and promise to respond last updated data
Pick Two
THECODEPROGRAM
A
P
Availability
AP
System can be distributed and promise to has high availability
Partition Tolerance

Theory of NOSQL: CAP

GIVEN:
- Many nodes
- Nodes contain *replicas of partitions* of the data

- Consistency
  - All replicas contain the same version of data
  - Client always has the same view of the data (no matter what node)
- Availability
  - System remains operational on failing nodes
  - All clients can always read and write
- Partition tolerance
  - multiple entry points
  - System remains operational on system split (communication malfunction)
  - System works well across physical network partitions

CAP Theorem:
satisfying all three at the same time is impossible



CAP theorem for NoSQL

What the CAP theorem really says:

- If you cannot limit the number of faults and requests can be directed to any server and you insist on serving every request you receive then you cannot possibly be consistent

How it is interpreted:

- You must always give something up: consistency, availability or tolerance to failure and reconfiguration

Task7: show number of fixed staff and their total average salary per gender [**you must use $divide**]

```
/*Task 7*/
db.employee.aggregate([
{ $match:{"Salary.Salary Type":"Fixed" }  },
{$group:{
        "_id":{"Gender":"$Gender"},
        "Number of Fixed Staff":{$sum:1},
        "Total Salary":{$sum:{$add:["$Salary.Net","$Salary.Housing"]}}
    }   },
{ $project:{
        "_id":0,
        "Gender":"$_id.Gender",
        "Number of Fixed Staff":1,
        "Total Salary":1,
        "Total Average Salary":{
            $divide:["$Total Salary","$Number of Fixed Staff"]
        } }
    }
])
```