**DASC 5333**

11/19/2024

**Introduction to concurrency control and transaction management**

by K. Yue

**1. Concepts of Concurrency Control**

* Modern databases are multi-user, multi-tasking systems: many users access the system concurrently with many tasks.
* A task may not be completed in one continuous execution. It may be divided into many execution steps.
* There are many concurrent tasks.
* There are no guarantees of the relative orders of concurrent tasks in an execution *schedule*.
* Without proper concurrency control,
  1. Read-write anomaly and write-write anomaly can occur.
  2. Database may become inconsistent.
* Task schedules need to maintain data and transaction integrity.

***Example:***

Use case: transfer $200 from account 1000 to account 2000.

-- Task t1  
-- Assumption: account 1000: $1,000, account 2000: $500  
-- Initial consistent state: total of two accounts: $1,500  
  
-- Step [1]:  
UPDATE Account SET amount=amount-200 WHERE account\_number=1000;  
  
-- Between step [1] and step [2]:  
-- Inconsistent state at this point; total of two accounts: $1,300  
  
-- Step [2]:  
UPDATE Account SET amount=amount+200 WHERE account\_number=2000; -- step [2]  
  
-- After completion of step [2]: consistent state again; total of two accounts: $1,500

Intended sequence #1 for task #1:

(1) Task t1 step [1]  
(2) Task t1 step [2]

Sequence #2: read-write anomaly.

(1) Task t1: step [1]  
(2) Task t2 reads the inconsistent state to produce an account report: account 1000: $800, account 2000: $500  
(3) Task t1: step [2]

Sequence #3: system crash and recovery

(1) Task t1: step [1]  
(2) System crashes; task t1 aborts after step [1]

Sequence #4: write-write anomaly.

(1) Task t1: step [1]  
(2) Task t2 reads and account amounts, calculate interest and update accounts. Interests will be calculated based on account 1000: $800, account 2000: $500.  
(3) Task t1: step [2]

**1.1 ACID Properties**

* Thus, to avoid accessing inconsistent states, concurrency control is necessary.
* Concurrency control is mainly done by transaction management.
* A transaction is a logical unit of database processing that is *atomic*: either the entire transaction is performed, or none of the transaction action is performed. This is the 'all or nothing' property.
* This refers to the famous ACID properties in DBMS: e.g. <http://en.wikipedia.org/wiki/ACID>
* ACID properties:
  1. Atomicity: A transaction is an atomic unit of processing. It is either performed in its entirety or not performed at all.
  2. Consistency preservation: A correct execution of a transaction must take the database from one physically consistent state to another. This is known as physical consistency.
  3. Isolation: A transaction should not make its updates visible to other tasks and transactions until it is committed.
     + In the example above, isolation disallows task t2 to access the inconsistent states of accounts 1000 and 2000 within task t1.
     + This property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
  4. Durability or permanency: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.
* Implementing ACID can bring performance degradation. Thus, for example, some NoSQL DB provide only 'eventual consistency'.
* Many NoSQL databases support the BASE model instead: Basic Availability, Soft-State, Eventual Consistency.
* Many DBMS support ACID by means of locking or multi-versioning.
* Basically, in locking, a transaction may have the exclusive access to selected data until the transaction is terminated.
* SQL support of transaction management depends on the vendor. It usually includes:
  1. The execution of a single SQL statement is atomic.
  2. The commands START TRANSACTION (or similar) and END TRANSACTION (or similar) are available to specify the boundary of transactions.
  3. COMMIT makes all data changes in the transaction to become permanent.
  4. ROLLBACK undoes all data changes in the transaction (or since the last COMMIT or ROLLBACK).

***Example:***

Use case: transfer $200 from account 1 to account 2000.

START TRANSACTION;  
  UPDATE Account SET amount=amount-200 WHERE account\_number=1000;  
  UPDATE Account SET amount=amount+200 WHERE account\_number=2000;  
  
IF ERRORS=0 COMMIT;  
IF ERRORS<>0 ROLLBACK;

**1.2 Application programmer's responsibility in terms of ACID**

1. Atomicity: the transaction is either fully committed, or fully rollback. The DB developer needs to define the scope and action of the transaction. (What is the transaction?)
2. Consistency: the execution of transaction should keep data consistent. It is the programmer's responsibility to ensure logical consistency: that the logic of the code is consistent with the problem requirements.

***Example:*** The following transaction can be atomic but *inconsistent*.

START TRANSACTION;  
  UPDATE Account SET amount=amount-200 WHERE account\_number=1000;  
  UPDATE Account SET amount=amount+400 WHERE account\_number=2000;  
  
IF ERRORS=0 COMMIT;  
IF ERRORS<>0 ROLLBACK;

1. Isolation: As a result, concurrent access will leave the database consistent. Usually not a concern for the application programmers.
2. Durability: Once committed, the transaction is finalized. Usually no concern for the application programmers.

**2. MySQL Transaction Management**

* MySQL Manual on TM statements: <https://dev.mysql.com/doc/refman/8.1/en/sql-transactional-statements.html>
* Autocommit mode: after the execution of a SQL statement, the result is automatically committed.
* START TRANSACTION disables the autocommit mode.
* MySQL supports COMMIT and ROLLBACK.
* It also supports LOCK TABLES and UNLOCK TABLES.

***Example:***

This is not a realistic example but it shows you an example of transaction management.

Suppose we have an ActiveStudent table on top of the Student table in toyu. The column numCourses is the number of courses a student has enrolled in. It is a derived column obtained by counted the number of classId the student in the enroll table.

CREATE TABLE IF NOT EXISTS activeStudent(  
    stuId        INT NOT NULL,  
    fname        VARCHAR(30) NOT NULL,  
    lname        VARCHAR(30) NOT NULL,  
    numCourses   INTEGER(4) DEFAULT 0  
);  
  
SELECT \* FROM activeStudent;  
  
-- Populating activeStudent initially.  
INSERT INTO activeStudent(stuId, fName, lName, numCourses)  
SELECT s.stuId, s.fName, s.lName, COUNT(e.classId) as numCourses  
FROM Student AS s LEFT JOIN Enroll AS e ON (s.stuId = e.stuId)  
GROUP BY s.stuId, s.fName, s.lName;  
  
SELECT \* FROM activeStudent;  
  
When we add the enrollment (100000, 10006, NULL, 0), we need to perform two tasks:

1. insert the row (100000, 10006, NULL, 0) into enroll.
2. increment numCourses for student 100000 by 1 in activeStudent

We can write a procedure to do so:

DROP PROCEDURE IF EXISTS enroll;  
  
DELIMITER //  
  
CREATE PROCEDURE enroll  
   (IN stuid VARCHAR(6),  
    IN classId VARCHAR(8),  
    IN grade VARCHAR(2),  
    IN n\_alerts INT)  
BEGIN  
  
DECLARE EXIT HANDLER FOR SQLEXCEPTION  
BEGIN  
   ROLLBACK;  
   RESIGNAL; -- pass on the error with no change.  
END;  
  
START TRANSACTION;  
  
INSERT INTO enroll  
VALUES (stuid, classId, grade, n\_alerts);  
  
UPDATE activestudent AS a  
SET a.numCourses = a.numCourses + 1  
WHERE a.stuid = stuid;  
  
COMMIT;  
  
END //  
  
DELIMITER ;  
  
SELECT \* FROM Enroll;  
SELECT \* FROM ActiveStudent;  
  
CALL enroll(100000, 10006, NULL, 0);  
CALL enroll(100009, 10006, NULL, 0);  
  
SELECT \* FROM Enroll;  
SELECT \* FROM ActiveStudent;  
  
DROP TABLE ActiveStudent;  
DROP PROCEDURE enroll;  
  
DELETE FROM enroll WHERE stuId = 100000 AND classId = 10006;  
DELETE FROM enroll WHERE stuId = 100009 AND classId = 10006;  
  
SELECT \* FROM Enroll;

**NoSQL Databases**

by K. Yue

**1. Introduction**

* NoSQL:
  + “Not Only SQL” or “Not SQL.” Relatively new kinds of non-relational database systems.
  + A loose category of many diverse database systems with different data models.
* Some common NoSQL features:
  + Non-relational
  + Distributed
  + High scalability
  + High availability
  + High performance and parallelism
  + No or flexible schema
  + Weaker support of ACID properties
  + Oriented towards semi-structured or non-structured data.
  + Weaker support of data integrity and constraints.
  + Usually more object-oriented.
  + More analysis-oriented than transaction-oriented.
  + Simpler APIs for database interaction.
* Major non-relational database systems as in top db-engines ranking, <https://db-engines.com/en/ranking> (rankings as of 9/1/2023):
  + Document models: e.g. *MongoDB* (rank #5), CouchBase (32), Firebase Realtime Database (38), CouchDB (44), Realm (56)
  + Key-value models: e.g. Redis (6), Memcached (33), etcd (52), HazelCast (55), LevelDB (111).
  + Wide-column models: e.g. Cassandra (12), Hive (17), HBase (26), Datastax Enterprise (60)
  + Graphical models:*Neo4J (22)*
* Some observations:
  + Search engines (not really a data model): ElasticSearch (8; document model), Splunk (14; key-value), Solr (24; document model)
  + Still dominated by relational databases.
  + Many DB support multiple models.
* Read:
  + A simple introduction to NoSQL database: <https://www.guru99.com/nosql-tutorial.html>
  + Introduction to NoSQL: <https://www.geeksforgeeks.org/introduction-to-nosql/>

**2. Advantages and Disadvantages**

* Some advantages of NoSQL databases in general:
  1. Distributed
  2. High scalability: horizontal scalability, as opposed to vertical scalability.
     1. Horizontal scalability (scaling out): add more machines to the distributed system.
     2. Vertical scalability (scaling up): add more power to existing machines; replace existing machines by more powerful one.
  3. High availability
  4. High performance
  5. Flexibility: flexible schema or schemaless
  6. More object-oriented:
     1. Better abstraction model
     2. Better interoperability with programming language. No need of object-relational mapping.
* Some disadvantages of NoSQL databases in general:
  1. Weaker data integrity support
  2. Weaker transaction support
  3. Weaker theoretical and design methodology support
  4. Relative lack of standards
  5. Relative lack of tools and interoperability
  6. Complexity

**3. ACID versus BASE Transaction Models**

**3.1 ACID**

* Relational database support ACID properties to support data consistency and integrity of transactions under concurrent access: e.g., <http://en.wikipedia.org/wiki/ACID>
* ACID properties (review):
  1. Atomicity: A transaction is an atomic unit of processing. It is either performed in its entirety or not performed at all.
  2. Consistency preservation: A correct execution of a transaction must take the database from one physically consistent state to another. This is known as physical consistency.
  3. Isolation: A transaction should not make its updates visible to other tasks and transactions until it is committed.
  4. Durability or permanency: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.
* Supporting ACID limits other desirable features: scalability, availability, and performance.

**3.2 BASE**

* To enhance scalability, availability and performance, most NoSQL DB do not fully support ACID.
* NoSQL supports different transaction models.
* The Basic Availability, Soft-State, Eventual Consistency (BASE model) for distributed database is the most popular one.
  1. Basic available: data is basically available across nodes of the distributed database, despite network failures.
  2. Soft-state: There is no immediate consistency. As a result, different replicas may have different values across the distributed systems at a given time. Thus, the state of the database is soft.
  3. Eventual consistency: eventually, data replicas will have the same value across the distributed database.

**3.3 CAP Theorem**

* Since NoSQL databases are mostly distributed, it is important to have some understanding of the famous CAP theorem for distributed data stores.
* See, for example, <https://en.wikipedia.org/wiki/CAP_theorem>.
* There are three *desirable* guarantees of *distributed* data stores, CAP:
  1. Consistency: the return value is always the same for the same data across the distributed systems.
  2. Availability: every request will return a response, either the data or an error. (Note that the return data may or may not always be the same).
  3. Partition tolerance: the database continues to operate in case of network partitions (one partition of the network cannot communicate with another partition because of message drops).
* The CAP theorem states that any distributed database can provide only two out of the three guarantees.
* Different databases based their designs on prioritizing two out of the three C-A-P.
* For a more detailed discussion, one may see:  <https://www.instaclustr.com/blog/cassandra-vs-mongodb/> (optional read):
  1. It contains a discussion how Cassandra and Mongo trade-off CAP.
  2. It also includes a discussion of a more refined CAP theorem: PACELC Theorem: "PACELC is summarized as follows: In the event of a partition failure, a distributed system must choose between Availability (A) and Consistency (C), else (E) when running normally it must choose between latency (L) or consistency (C)."

**4. Major NoSQL data models**

**4.1 Key-value model**

1. Data is stored as key-value pairs.
2. Values can possibly be JavaScript Object Notation (JSON) strings, which store serialized objects.
3. Some key-value databases support rich JSON queries.

**4.2 Document Model**

1. Document-oriented databases store data as documents.
2. Documents can be considered as semi-structured data.
3. Thus, XML databases can be considered as employing the document model.
4. Modern document-oriented databases commonly employ JSON. E.g., MongoDB and CouchDB.
5. The document model can be considered as a subclass of key-value model.
   1. The stored value of a key-value model can be a document.
   2. The stored value can be manipulated by operations based on the selected document model (mostly JSON).
6. MongoDB is likely the most popular document-oriented NoSQL DB. It will be covered in more details in this class.

***Example:*** In CouchDB, a key-value pair may be:

**Key:**"MBSEBaseModel~939c7672-5d2d-11ec-bf63-0242ac130002"  
  
**Value to be stored:**

{  
   "BCAssetId": "939c7672-5d2d-11ec-bf63-0242ac130002",  
   "BCAssetType": "MBSEBaseModel",  
   "BCAssetName": "Gateway-PPE-Base-Model",  
   "BaseModelDesc": "PPE project's model.",  
   "version": {  
      "version": "2.1",  
      "subversion": "4.6",  
      "startTime": "2021-12-08T17:25:23+06:00"  
   },  
   "storage": {  
      "isEncrypted": true,  
      "EncrypMethod": "AES256",  
      "EncrypKey": "q4t7w!z%C\*F-JaNdRgUkXp2r5u8x/A?D",  
      "useIPFS": true,  
      "IPFSCid": "QmT5NvUtoM5nWFfrQdVrFtvGfKFmG7AHE8P34isapyhCxX",  
      "IPFS\_HashHead": "A0Xa",  
      "payloadRaw": "raw PPE MBSE Base model description. V2.1.4.6.",  
   }  
}

Note:

* CouchDB adds two fields, \_id and \_rev, automatically if they are not supplied.
* The field \_rev is used for multi-version concurrency control (MVCC) to ensure 'eventual consistency.'
* Some may consider CouchDB as a document-oriented database. The boundary between key-value model and document model is not clear cut.

To query CouchDB, one may use many methods. Examples:

1. CouchDB RESTful API: <https://docs.couchdb.org/en/latest/api/index.html>
2. MapReduce-based views: <https://docs.couchdb.org/en/latest/ddocs/views/index.html>
3. Mango query

An example Mango query that returns all CouchDB key-value pairs for MBSEBaseModel.

{  
    "selector": {  
        "BCAssetType": { "$eq": "MBSEBaseModel" }  
    }  
}

See <https://docs.couchdb.org/en/latest/api/database/find.html> for more information about selector syntax.

**4.3 Wide-Column Model**

1. A columnar DBMS or column-oriented DBMS stores data tables grouped by columns, instead of grouped by rows (as in most relational DBMS).
   1. For example, related columns may be stored together in a file for faster performance.
   2. Benefits:
      1. Faster access for certain types of queries.
      2. Better chance for data compression.
   3. Disadvantages:
      1. Slower update.
      2. Slower access for certain types of queries.
2. Read introductions to the wide-column model. Examples:
   1. <https://dandkim.com/wide-column-databases/>
3. In wide column model, data is stored as keys and columns. Each column contains a column-name and a value.
4. Thus, to get a data value, use (key, column-name).
5. Cassandra is one of the most popular wide-column databases.

**4.4 Graphical Model**

1. "A graph database stores nodes and relationships instead of tables, or documents."
2. Quite object-oriented, using a directed graph model.
3. neo4j is the most popular graphical database.
4. Introduction: <https://neo4j.com/developer/graph-database/>.
5. To start learning Neo4j, download and install Neo4j desktop.
6. Neo4j Query Language: Cypher, <https://neo4j.com/developer/cypher/>.
7. Basic Cypher syntax: (nodes)-[:ARE\_CONNECTED\_TO]->(otherNodes).

**Introduction to MongoDB**

by K. Yue

**1. Introduction**

* NoSQL document model distributed database owned by MongoDB (NASDAQ: MDB).
* Documents are stored in JSON format.
* Three versions:
  + Community server: open source and version
  + Enterprise server: commercial version
  + Atlas: cloud version

**1.1 Installation**

For this class, install the followings.

1. MongoDB community server: ensure that it includes Mongo Compass, a MongoDB client, <https://www.mongodb.com/try/download/community>
2. Mongo Shell (client)
   1. mongosh.exe: a Javascript shell for interacting with MongoDB, <https://www.mongodb.com/try/download/shell>.
   2. Do *not* use mongo.exe, the deprecated former shell.
3. Mongo Compass includes a Mongosh.
4. A screenshot of a computer

   Description automatically generated
5. MongoDB tools: command line utilities including import and export, <https://www.mongodb.com/try/download/database-tools>.
   1. After unzipping, you may put mongosh and these utilities in the same location of the other mongoDB programs, e.g., C:\Program Files\MongoDB\Server\5.0\bin.
   2. You may add the directory “C:\Program Files\MongoDB\Server\5.0\bin”, or similar, in the system PATH variable so these tools can be used anywhere.
6. To be able to use MongoDB through Python, you will to install a driver: "pip install pymongo" in cmd.

**1.2 Server-Client DBMS architecture**

* Like many DBMS, MongoDB uses a client server model.
* Server:
  + In case the MongoDB server has not been started, run "mongod" in a command terminal.
  + To check whether mongod is running, execute 'tasklist /FI "IMAGENAME eq mongod.exe"' in Command CLI.
  + It listens to a port to accept and interpret commands and return results.
  + mongod's default port: 27017.
* Clients: send MongoDB commands and accept results. Clients used in this course:
  + Mongo Compass
  + mongosh
  + Python through pymongo (if Python is used.)

**1.3 Resources**

* MongoDB manual: <https://docs.mongodb.com/manual/>

**2. MongoDB Structures**

* MongoDB is structured as db -> collection -> document (object/JSON; nested structures) in a way similar to db -> table (flat structure) -> row in relational DB.
* Thus, documents are inserted into a collection of a db.
* db and collection do not need to exist before referencing them.
* In MongoDB's db, within mongosh:
  + 'use tinker' set the default db to tinker.
  + The keyword db refers to the default db.
  + If 'tinker' does not exist, it will be created.

**2.1 Using mongo command CLI through mongosh**

* Run 'mongosh' in command CLI in your working directory.
* Mongosh accept JavaScript commands in a mongo shell setting.
* For inserting documents, it supports two methods, insertOne and insertMany.
* See mongosh CRUD:<https://docs.mongodb.com/mongodb-shell/crud/insert/>.

**3. Writing to Mongo**

1. See CRUD operation in Mongo Guide to begin with: <https://docs.mongodb.com/guides/>.
   1. However, the guide uses the deprecated shell "mongo" instead of "mongosh".
   2. Since mongosh should be used, be mindful of discrepancies.

***Example:***

In mongosh, execute the code:

use tinker  
db.test1.insertOne(  
   {  
      "StudentId" :1,  
      "StudentName" : "Joseph Connor"  
   }  
)  
  
gives the following result:

test> use tinker  
switched to db tinker  
tinker> db.test1.insertOne (  
...     {  
.....           "StudentId" :1,  
.....           "StudentName" : "Joseph Connor"  
.....   }  
... )  
{  
  acknowledged: true,  
  insertedId: ObjectId("61e0d5f36753d9628bb4bfa1")  
}  
tinker> db.test1  
tinker.test1

Note:

1. In "db.test1.insertOne (", the '(' must not be put into the next line.
2. If not, mongosh thinks that the current JavaScript statement has ended and you may get:

tinker>   
[Function: insertOne] AsyncFunction {  
  apiVersions: [ 1, Infinity ],  
  serverVersions: [ '3.2.0', '999.999.999' ],  
  returnsPromise: true,  
  topologies: [ 'ReplSet', 'Sharded', 'LoadBalanced', 'Standalone' ],  
  returnType: { type: 'unknown', attributes: {} },  
  deprecated: false,  
  platforms: [ 0, 1, 2 ],  
  isDirectShellCommand: false,  
  acceptsRawInput: false, db.test1.insertOne  
  shellCommandCompleter: undefined,  
  help: [Function (anonymous)] Help  
}  
tinker> (  
...     {  
.....           "StudentId" :1,  
.....           "StudentName" : "Joseph Connor"  
.....   }  
... )  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
  
In Windows, you may start Compass through the startup manual:

A screenshot of a computer

Description automatically generated

In Mongo Compass (you may enter nothing in the 'Paste your connection string' connect box):

A screenshot of a computer

Description automatically generated

* Note that a field \_id with a system generated object id is created. It is unique within the db and can be served as an id.

If the code is executed one more time, Mongo Compass has:

A screenshot of a computer

Description automatically generated

Note:

1. There are now two Joseph Connor.
2. StuId is not a 'primary key'.
3. Document model is not set-theoretic. Relation model is set-theoretic.

To insert a document 'doc' only when it does not already exist, use something like:

if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }

Note:

1. 'db.test1.find(doc)' finds the documents doc (one document in the example below). It returns a cursor, which is an iterator of the query result.
2. cursor has a method count() to count the result.

The following session illustrates this concept.

Code:

show dbs  
db.dropDatabase()  
show dbs  
  
// remove tinker  
use tinker  
db.test1.find()  
doc = {  
      "StudentId" :1,  
      "StudentName" : "Joseph Connor"  
}  
doc  
if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
db.test1.find()  
if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
db.test1.find()

Session:

tinker> db.test1.find()  
  
tinker> doc = {  
...             "StudentId" :1,  
...             "StudentName" : "Joseph Connor"  
... }  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> doc  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
{  
  acknowledged: true,  
  insertedId: ObjectId("61e0e49e6753d9628bb4bfa5")  
}  
tinker> db.test1.find()  
[  
  {  
    \_id: ObjectId("61e0e49e6753d9628bb4bfa5"),  
    StudentId: 1,  
    StudentName: 'Joseph Connor'  
  }  
]  
tinker> if (db.test1.find(doc).count() == 0) { db.test1.insertOne(doc) }  
  
tinker> db.test1.find()  
[  
  {  
    \_id: ObjectId("61e0e49e6753d9628bb4bfa5"),  
    StudentId: 1,  
    StudentName: 'Joseph Connor'  
  }  
]

**3.2 Unique Index**

* A unique index can be used to ensure that all documents within the collection must have unique values on the fields.
* This can be used for use cases of inserting the document only if the unique index has an unique value.
* Thus, a unique index can serve as a candidate key (if *it is not missing*) for identifying document in the collection.

***Example:***

Code:

// remove tinker  
show dbs  
db.dropDatabase()  
show dbs  
// create index  
db.test1.createIndex( { "StudentId": 1 }, { unique: true } )  
doc = {  
      "StudentId" :1,  
      "StudentName" : "Joseph Connor"  
}  
doc  
db.test1.insertOne(doc)  
db.test1.insertOne(doc)  
  
  
Session:

tinker> // create index  
  
tinker> db.test1.createIndex( { "StudentId": 1 }, { unique: true } )  
StudentId\_1  
tinker> doc = {  
...       "StudentId" :1,  
...       "StudentName" : "Joseph Connor"  
... }  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> doc  
{ StudentId: 1, StudentName: 'Joseph Connor' }  
tinker> db.test1.insertOne(doc)  
{  
  acknowledged: true,  
  insertedId: ObjectId("6570fb99629ad72db73f7bcf")  
}  
tinker> db.test1.insertOne(doc)  
MongoServerError: E11000 duplicate key error collection: tinker.test1 index: StudentId\_1 dup key: { StudentId: 1 }

Note:

* In 'db.test1.createIndex( { "StudentId": 1 }, { unique: true } )', '"StudentId": 1' means the attribute is a part of the index. It does not mean the value of "StudentId" should be one. 1 stands for true here.
* In { unique: true }, the index is set to have the uniqueness property.

***Example:***

db.test1.insertMany([  
   {   "StudentId" :2,  
      "GPA": 3.72  
   },  
   {   "StudentId" :3,  
      "GPA": 1.69  
   },  
   {  
      "BCAssetId": "78c22fc6-5dec-11ec-bf63-0242ac130002",  
      "BCAssetType": "BCAssetTypeMetadata",  
      "BCAssetName": "BCAssetTypeMetadata: MBSEModel",  
      "ForBCAssetType": "MBSEModel",  
      "Version": {  
         "Version": "1.0",  
         "Subversion": null,  
         "StartTime": "2019-01-13T07:23:13+06:00"  
      }  
   }  
])  
db.test1.find()

Note:

1. The method insertMany() inserts many documents.
2. Documents may have *no*schema.
3. Within a collection, there can be many kinds of documents.
4. StudentId is a unique index, but it may not exist.
5. Thus, a Mongo's unique index is not exactly the same as a candidate key (which cannot be null) of a table in the relational model.

**4. Querying**

* Basically use the find method.
* find as supported in Mongosh: <https://docs.mongodb.com/manual/reference/method/db.collection.find/>.
* Format: db.collection.find(query, projection).

**4.1 Toyu**

Create the ‘toyu’ database in MongoDB.

1. Download the file: [toyu-db.gz](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/nosql/toyu-db.gz).
2. Ensure that you have download MongoDB tools: command line utilities including import and export, <https://www.mongodb.com/try/download/database-tools>.
3. Run the command:

mongorestore --archive="toyu-db.gz" --gzip --nsFrom='toyu.\*' --nsTo='toyu.\*'

Note that the design of toyu is not the typical way one would design a MongoDB. Instead, it is intended to look like the toyu MySQL database for ease of comparison.

***Example:***

[1] Show all students.

use toyu  
db.student.find()

Getting rid of \_id:

db.student.find({},  
   { "\_id": 0 }    
)

[2] // Show all information of students majoring in 'CINF'.  
  
db.student.find({"major": "CINF"},  
    { "\_id": 0 }  
)  
  
[3] Show all student names. Return an array of student objects.

db.student.find({},  
   { "fname": 1, "lname":1, "\_id": 0 }    
)

[4] Show all student names in this format:

student #0: Tony Hawk  
student #1: Mary Hawk  
student #2: David Hawk  
student #3: Catherine Lim  
student #4: Larry Johnson  
student #5: Linda Johnson  
student #6: Lillian Johnson  
student #7: Ben Zico  
student #8: Bill Ching  
student #9: Linda King

Solution:

result = db.student.find({},  
   { "fname": 1, "lname":1, "\_id": 0 }    
).toArray()  
  
// May not always work as toArray() returns a promise,  
// which may not be ready for use.  
result.forEach((x,i) => console.log('student #' + String(i) + ': ' + x["fname"] + ' ' + x["lname"]))

[5] Show the names and credits (ach) of students majoring in 'CSCI' and having 40 or more credits.

db.student.find(  
   { "major": "CSCI", "ach" : {$gte: 40} },  
   { "fname": 1, "lname":1, "ach":1, "\_id": 0 }    
)

Notes:

1. MongoDb's query and projection operators: <https://docs.mongodb.com/manual/reference/operator/query/>

[6] Show the first name and last name of students with a first name starting with a L or B, case insensitive.

db.student.find(  
   { "fname": { $regex: /^[lb]/, $options: "i" } },  
   { "fname": 1, "lname":1, "\_id": 0 }    
)

Notes:

1. A regular expression is used: <https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex>.
2. For regular expressions in general, see: <https://en.wikipedia.org/wiki/Regular_expression>
3. Explanations:
   1. ^: match the beginning of a string.
   2. [lb]: a character class that matches 'l', 'b' (and also 'L' and 'B' since case insensitive matching is used.)
   3. option a: case insensitive matching.

[7] Show the names and credits (ach) of students majoring in 'CSCI' and having 40 or more credits.

db.student.find(  
   { "$and": [ { "major": "CSCI"}, { "ach": {"$gte": 40}} ] },  
   { "fname": 1, "lname":1, "ach":1, "\_id": 0 }    
)

**4.2 Aggregation**

1. "Aggregation operations process multiple documents and return computed results."
2. See: <https://docs.mongodb.com/manual/aggregation/>.
3. It can be used to replace map-reduce functionality. See: <https://docs.mongodb.com/manual/reference/map-reduce-to-aggregation-pipeline/>.
4. There will not be programming questions on aggregation in the final examination.

[8] Show the number of faculty in each department.

In SQL:  
  
SELECT DISTINCT deptCode, Count(facId)  
FROM faculty  
GROUP BY deptCode;

In MongoDB:

db.faculty.aggregate([  
    {"$group" : {\_id:"$deptCode", "count":{$sum:1}}}  
])  
  
db.faculty.aggregate(  
   [    
      { $group: { "\_id": "$deptCode", "count": {$sum:1}} },  
      { $project: { "deptCode": "$\_id" , "num\_faculty": "$count",  "\_id": 0}}  
   ]  
)

Notes:

1. $group: form group.
2. $sum: aggregate function.

[9] Show the names of students who have enrolled in 10000: joining two document.

This should have the similar effect of the SQL statement:

SELECT DISTINCT s.fname, s.lname  
FROM student AS s, enroll AS e  
WHERE s.stuId = e.stuId AND e.classId = 10000;

In MongoDB:

db.student.aggregate([  
{$lookup:  
    {  
      from: "enroll",  
      let: {joinValue: '$stuId'},  
      pipeline: [  
           { $match:  
                 { $expr:  
                    { $and:  
                       [  
                         { $eq: [ "$stuId",  "$$joinValue" ] },  
                         { $eq: [ "$classId", 10000 ] }  
                       ]  
                    }  
                 }  
            }      
        ],  
        as: "enrollment"     }},  
  { $match: {"enrollment":  { $ne: [] }}},    
  { $project: { "fname": 1, "lname": 1, "\_id": 0}}   
])

Notes:

1. An 'join' example.
2. Joining is difficult in MongoDB than SQL as document database should not be designed like a relational database.
3. In particular:
   1. The relational model uses a flat structure with no embedment.
   2. The document model uses a hierarchical structure encouraging embedment.

**4.3 Running Javascript program not using mongosh**

Try run [tinker.js.txt](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/nosql/tinker.js.txt) (remove .txt when saving)

// run "npm i mongodb" in the working directory.  
  
// To run this program: node tinker1.js  
const mongo = require('mongodb');  
  
var MongoClient = mongo.MongoClient;  
var url = 'mongodb://localhost:27017';  
  
MongoClient.connect(url, function(err, client) {  
   db = client.db("toyu");  
   console.log("hello");  
   var result = db.collection("faculty").find(  
      { "rank": "Assistant Professor" },  
      { "fname": 1, "lname": 1, "deptCode": 1, "\_id": 0,  }    
   ).toArray()  
   result.then((docs) => {  
        console.log(docs);  
    }).catch((err) => {  
        console.log(err);  
    }).finally(() => {  
        client.close();  
    });  
});