Scalability
Database schema

```
CREATE TABLE IF NOT EXISTS
Account(
  account_id  CHAR(20)    PRIMARY KEY,
  balance     REAL        NOT NULL
);
CREATE TABLE IF NOT EXISTS
Transaction(
  id              SERIAL      PRIMARY KEY,
  to_account      CHAR(20)    NOT NULL,
  from_account    CHAR(20)    NOT NULL,
  amount          REAL        NOT NULL,
  tags            TEXT[]
);
CREATE TABLE IF NOT EXISTS
Tag(
  id      SERIAL  PRIMARY KEY,
  content TEXT    UNIQUE NOT NULL
);
CREATE TABLE IF NOT EXISTS
Transaction_Tag(
  transaction_id  INT     REFERENCES Transaction(id),
  tag_id          INT     REFERENCES Tag(id),
  PRIMARY KEY (transaction_id, tag_id)
);
```

Transaction_Tag is a table for many-to-many relationship between transactions and tags. It references Transaction and Tag’s id. Each row is a pair of one transaction’s id and one tag’s id. Each pair indicates the transaction has the corresponding tag. This table is mainly for tag search in Query request.

The text array field “tags” in table Transaction is only for response construction. It will be much faster than doing queries to retrieve corresponding tags from Transaction_Tag table. It won’t be used as tag search in Query request.

Create index

```
CREATE UNIQUE INDEX IF NOT EXISTS account_id ON Account(account_id);
CREATE INDEX IF NOT EXISTS balance ON Account(balance);
CREATE UNIQUE INDEX IF NOT EXISTS transaction_id ON Transaction(id);
CREATE INDEX IF NOT EXISTS amount ON Transaction(amount);
CREATE UNIQUE INDEX IF NOT EXISTS tag_id ON Tag(id);
```

Possible search methods are account_id-based, balance-based, amount-based and tag-based. Tag-based search can be transferred into tag_id-based search. Also when a tag's corresponding transactions are retrieved, it retrieves set of transaction's id, which means search based on transaction's id is required. So 5 indexes shown above are created to accelerate search.

Handle request

First transfer string-typed XML into tree-typed XML. When we have the root of the tree, based on idea of Depth-first Search, we iterate through children of the root and see if one is "create", "transfer", "balance" or "query". We have 4 corresponding functions to handle the 4 situations.

For create, we check the format of XML structure, validation of new account (64-bit u_int, existence) and balance(float). Then we build up SQL query, with properly transferred account and balance value. At last, tag with proper information in the XML response tree is built.

For transaction, we check the format of XML structure, validation of source/target account (64-bit u_int, existence), amount(float) and tags(existence). Then we build up SQL query, with properly transferred source/target account and amount value.

Specially mentioned, the process of adding transaction is done in the lock to make sure balance isn't modified and account isn't cleaned by other threads. Otherwise, balance may be withdrawn from source account but the target account has been reset during this time. Or balance is withdrawn from 2 transactions at the same time, but the balance is insufficient for either one.

Inside the lock, we first check if the target account exists. Then we check and update source account balance. At last we update target account's balance. As long as one step fails, the subsequent steps won't be executed. The whole transaction will fail, and previous execution won't be committed. Only all execution success will the transaction be committed. Then the lock is released.

For tags, we first check table Tag to see if the tag exists. If not, we first add the tag into table Tag. Then for all tags in the transaction, we retrieve a set of their id. We then use the set to build up rows in table Transaction_Tag.

At last, tag with proper information in the XML response tree is built.

For balance, simply the SQL query is build and use the query result to build tag in the XML response tree.

For query, we first use Depth-first Search to build up query condition sentence. We iterate through the XML request tree, and dive into next level if "and", "or", "not" are encountered. We return string based on "equals", "less", "greater" if "from", "to", "amount" are encountered. After all children are iterated, the "and", "or", "not" level assembles all return values with corresponding "and", "or", "not" tag. So it's a post-order Depth-first Search.

For tag-based search in the query, we build up SQL sentence to query table Transaction_Tag and Tag where tag's content is the given string, then retrieve a set of transaction_id linked with the selected tag_id. We then use the set to build query like "Transaction.id = transaction_id_set[0] OR Transaction.id = transaction_id_set[1] OR ..." and return to upper level.

For any format error in XML request tree, we set an error flag and return empty string. If the flag is set, the SQL query is aborted, else we use the query result to build XML response tree.

Muitithread

After the socket binds and starts listening to the port, we pre-create 256 threads, sending the socket as argument into listen_multiple function to wait for connection. It means at the same time there are at most 256 threads waiting on the same port. When one connection is built, any waiting thread can take the request and handle it in handle_request function. When response is finished, the thread returns to listen_multiple function, and is ready for next connection.

```
def main()
   …...
   socket.bind((host, port))
   socket.listen(256)
   for i in range(256):
      t = threading.Thread(target=listen_multiple, args=(socket,))
      t.start()

def listen_multiple(s):
   while True:
      conn, addr = s.accept()
      handle_request(conn)

def handle_request(conn):
   length = conn.recv(8)
   xml_length = int(length.decode("utf-8"))
   …...
```

Performance Analysis:
when we test our server, we created multiple test cases. For each request, we sent **20 tags**, mixing with transfer, query, balance and create .At the beginning, we would send create account request to clean the origin stats in database and **create 500** accounts. We used two computers to send test cases at the same time and calculated the maximum **response time** as our results

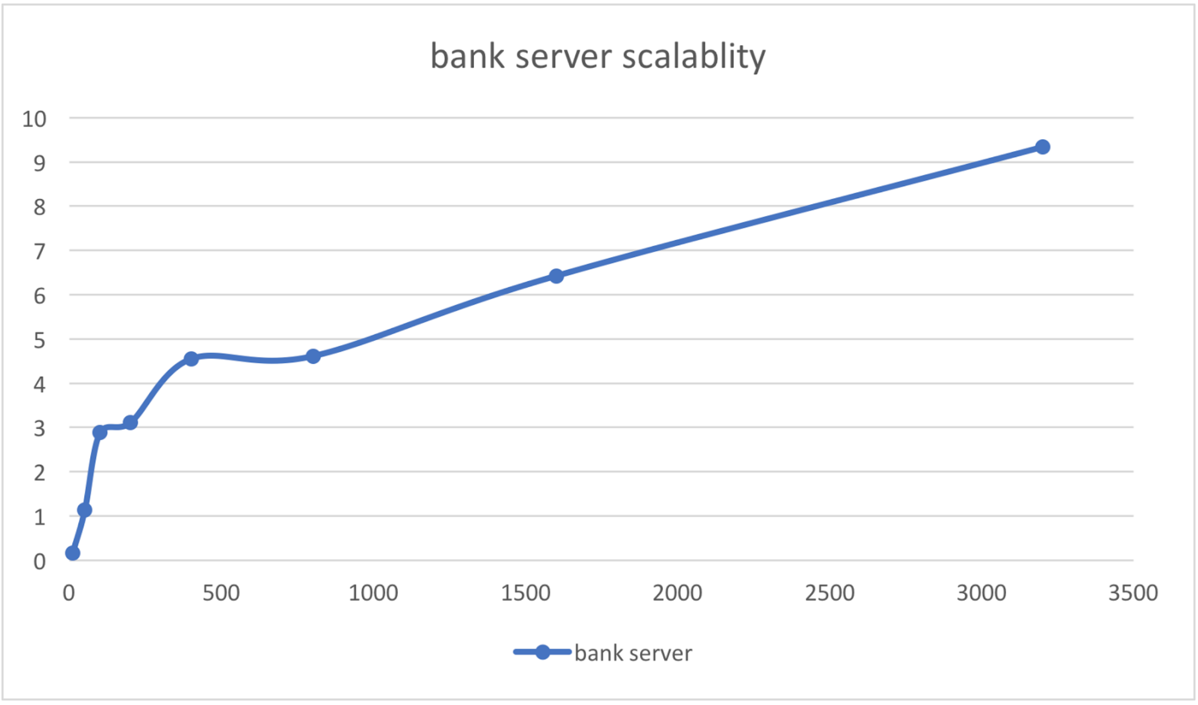


figure 1 server scalablity with competing request

Competing Requests	Response time(s)
10	0.167
50	1.14
100	2.88
200	3.11
400	4.55
800	4.61
1600	6.42
3200	9.33

table 1

For the figure 1, we can see the latency for the request did not increase in linearly. It's because we use the mulithreads pool to listen the 12345 port at the same time and create index to improve the database performance. When those request arrive the server port at the same time, the 256 threads in server will also accept those requests and handle 256 requests at the same time. Also with the help of database index and cache, the executing thread does not need to “actually” access database every time. So, we get the scalability in our bank_mt program

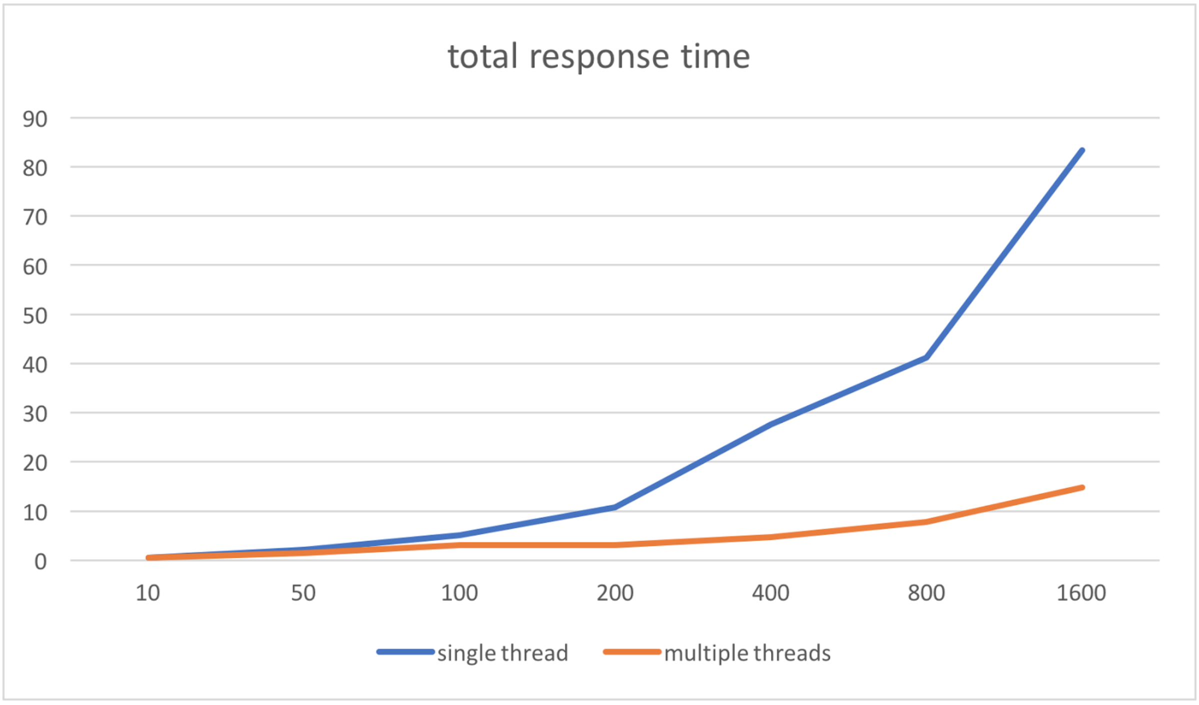
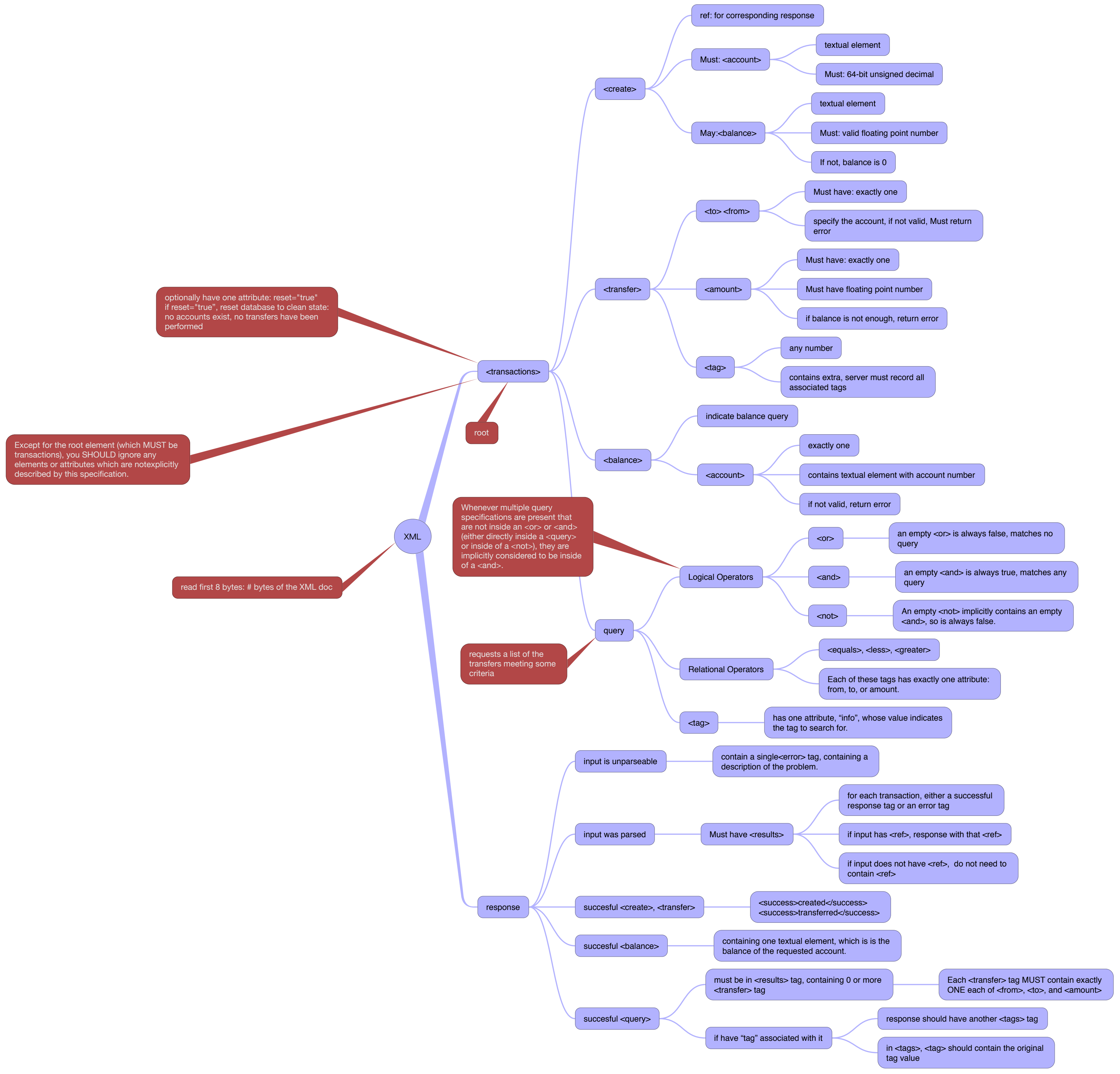


figure 2 compare with single thread and multiple threads

When we compare the single thread performance with multiple threads performance, the total response for single thread is almost linear with the increasement of request. It’s because the single thread server is always performing one thread and process such single task is almost consuming constant time. For the multiple threads part, we can get good scalability if the number of requests is below 800. It’s because of the limitation of hardware in our server. We could only use 256 threads to actively listen and handle the requests. If the number of requests is below 400, we almost handle those request “at the same time”. However, when this number exceeds the 400, we still only have 256 threads to handle requests, so the handling process could not be done “at the same time”, it still needs to multiple time to finish those requests.


XML
read first 8 bytes: # bytes of the XML doc
<transactions>
optionally have one attribute: reset="true" if reset="true", reset database to clean state: no accounts exist, no transfers have been performed
Except for the root element (which MUST be transactions), you SHOULD ignore any elements or attributes which are notexplicitly described by this specification.
root
<create>
ref: for corresponding response
Must: <account>
textual element
Must: 64-bit unsigned decimal
May:<balance>
textual element
Must: valid floating point number
If not, balance is 0
<transfer>
<to> <from>
Must have: exactly one
specify the account, if not valid, Must return error
<amount>
Must have: exactly one
Must have floating point number
if balance is not enough, return error
<tag>
any number
contains extra, server must record all associated tags
<balance>
indicate balance query
<account>
exactly one
contains textual element with account number
if not valid, return error
query
requests a list of the transfers meeting some criteria
Logical Operators
Whenever multiple query specifications are present that are not inside an <or> or <and> (either directly inside a <query> or inside of a <not>), they are implicitly considered to be inside of a <and>.
<or>
an empty <or> is always false, matches no query
<and>
an empty <and> is always true, matches any query
<not>
An empty <not> implicitly contains an empty <and>, so is always false.
Relational Operators
<equals>, <less>, <greater>
Each of these tags has exactly one attribute: from, to, or amount.
<tag>
has one attribute, "info", whose value indicates the tag to search for.
response
input is unparseable
contain a single<error> tag, containing a description of the problem.
input was parsed
Must have <results>
for each transaction, either a successful response tag or an error tag
if input has <ref>, response with that <ref>
if input does not have <ref>, do not need to contain <ref>
succesful <create>, <transfer>
<success>created</success>
<success>transferred</success>
succesful <balance>
containing one textual element, which is is the balance of the requested account.
succesful <query>
must be in <results> tag, containing 0 or more <transfer> tag
Each <transfer> tag MUST contain exactly ONE each of <from>, <to>, and <amount>
if have "tag" associated with it
response should have another <tags> tag
in <tags>, <tag> should contain the original tag value