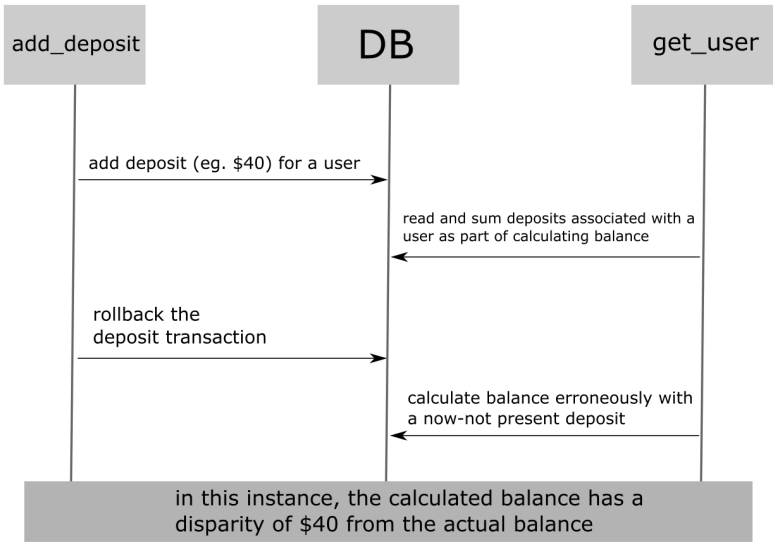
Concurrency Control For Top 3 Complex Transactions

Get User

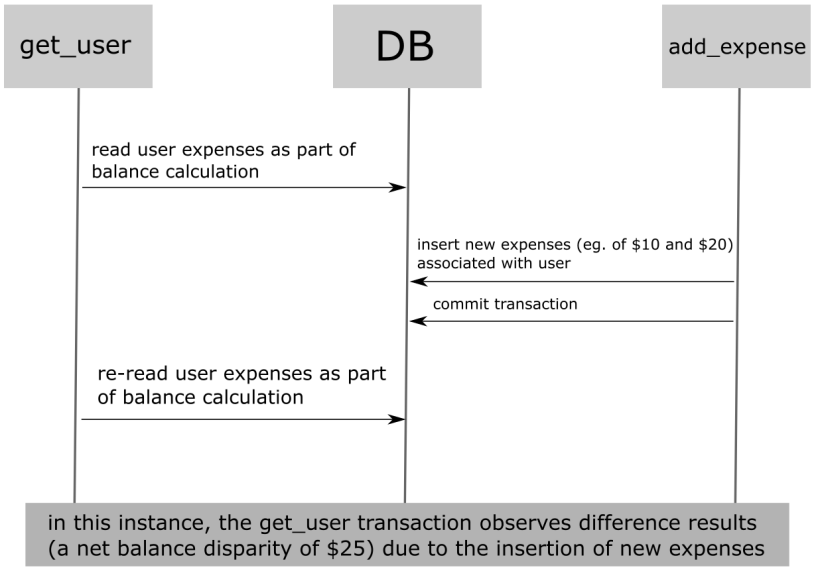
This endpoint gets information associated with a user - their name, username, and calculated net balance. The net balance present in a user's account is determined by the difference of the sum of deposits and the sum of expenses associated with the user; as such, the user's expenses and deposits are read.

Without concurrency control, the following cases could occur:

- Dirty Read: A dirty read could occur if either an expense or deposit associated with a user is added to its respective table (which get_user reads from) but is then rolled back. The example below uses the add_deposit endpoint.



- Phantom Read: A phantom read could occur should multiple expenses or deposits associated with a user be inserted into their associated tables while the get_user transaction is ongoing. The example below uses the add_expense endpoint.

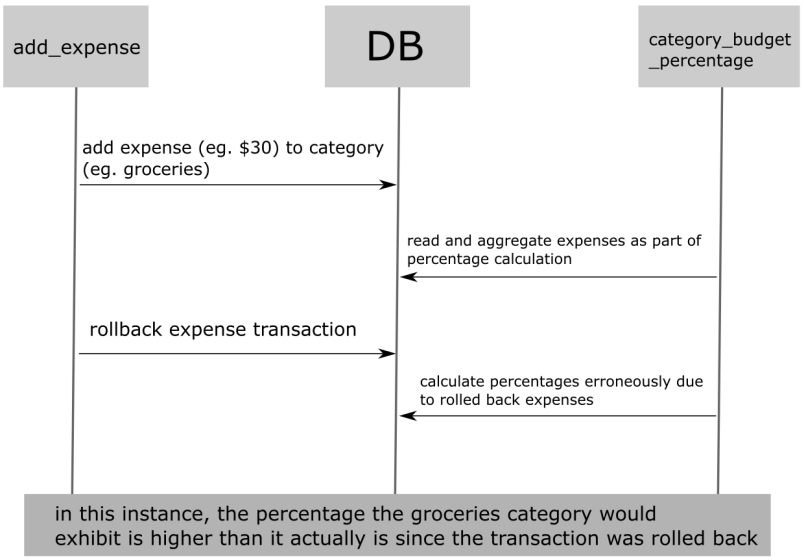


Category Budget Percentage

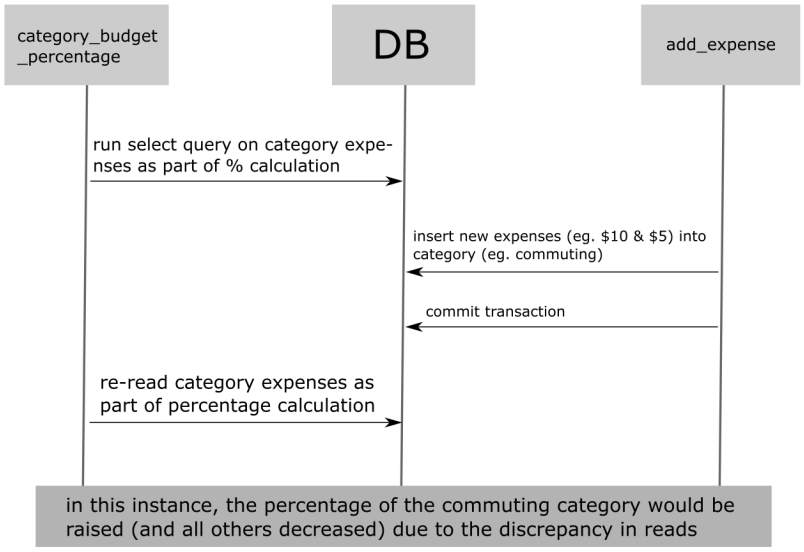
This endpoint determines the percentages of a user's total allocated budget on a per-category basis. The categories table is read, then the budget table is read, then the percentages are calculated based on the values gathered.

Without concurrency control, the following cases could occur:

- Dirty Read: A dirty read could occur when an expense associated with a category is added and then rolled back, meaning that the percentages would be skewed. This is represented in the following diagram:



- Phantom Read: A phantom read could occur when multiple expenses are inserted into a category while the category_budget_percentage transaction is still going on. In this instance, the same query would result in different percentages. This is represented in the following diagram:

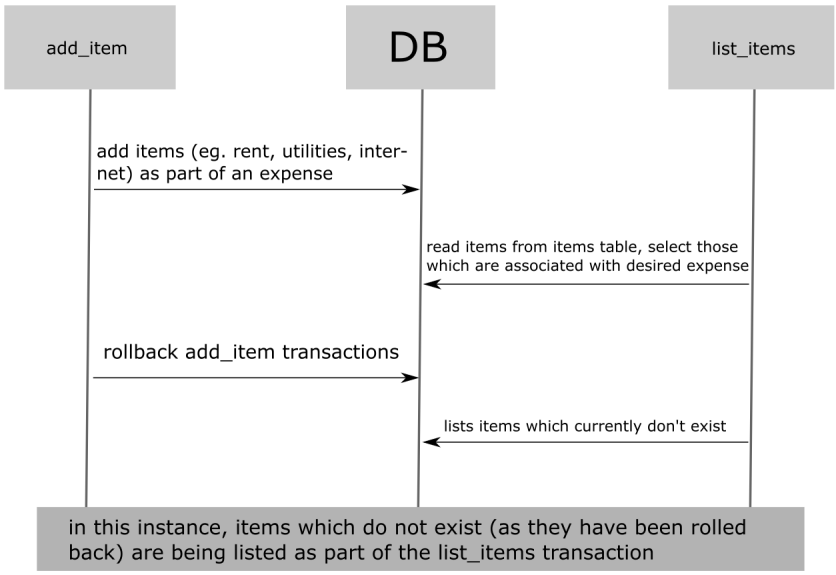


List Items

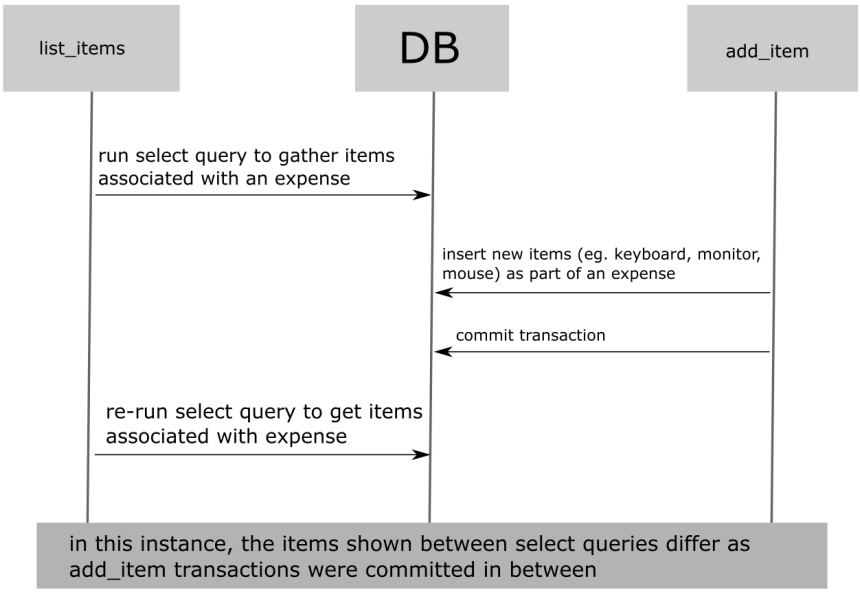
The list items endpoint lists the items associated with a particular expense. The expenses table is read to get the correct expense_id, followed by the items table to gather all the items associated with said expense.

Without concurrency control, the following cases could occur:

- Dirty Read: A dirty read could occur should an item be added that is associated with an expense, but the transaction is rolled back. This would result in an extra (now removed) item being shown in the list. This is represented in the following diagram:



- Phantom Read: A phantom read could occur should multiple items be added to an associated expense while the list items transaction is ongoing; in this instance, the query associated with list items would result in different rows being returned. This is represented in the following diagram:



Ensuring Proper Isolation

To ensure isolation of our transactions, we opted to go with the `REPEATABLE READ` isolation level (that is, using snapshot isolation). For one, as more secure levels of isolation prove increasingly time consuming, using snapshot isolation proves a good tradeoff between good concurrency control and efficient transactions. Given the scale and minimal user base of the project, the overhead associated with REPEATABLE READ is a fair cost to take compared to the speed of lower isolation levels. This also covers relevant read errors meaning that concurrency issues will be heavily minimized.
In addition, using this option proves far easier to implement as it is built-in to Postgres and requires no additional effort development-side (compared to manual implementation in each endpoint). This makes our code more readable and more easily understandable & modifiable, which also proves important should future revisions occur.