Transacciones

CI-0127 Bases de Datos, Universidad de Costa Rica

Sivana Hamer

Importante

Este documento recopila contenidos de diversos de sitios web especializados, académicos y documentos compartidos por universidades. Toda la información es utilizada con fines estrictamente académicos. Para más información, puede ver las referencias.

1 Introduction

It is common that when we execute operations in a database, from the point of view of a user it seems as a *single logical work unit*. However, frequently we require multiple operations to perform a work unit, requiring either *all* the work to be executed or *none at all* if there is a failure. Per example, if Alice wants to transfer $100 to Bob's bank account (logical work unit), the transfer requires to:

1. Check if Alice has enough funds ($100) to transfer to Bob.
2. Debit the $100 out of Alice's bank account.
3. Add the $100 to Bob's bank account.

Other examples of systems that use transactions include ticket reservation, and retail purchasing. To manage these types of operations, we need to perform the collection of operations as one logical work unit called a *transaction*.

2 Definition

A database operates over data items $A, B, C, \cdots$. A *data item* can be an attribute, tuple, page, disk block, table or database. Furthermore, we can perform the following operations:

- *read(X):* Reads the data item X from the database.
- *write(X):* Writes the variable X to a data item of the database. We asume that the writes are saving directly to disk, though that might is usually not the case as it is probably saved in cache or a buffer.

A *transaction* can be represented, in a simplified form, as a collection of $read(X)$ and $write(X)$. A transaction is delimited between a *begin transaction* and *end transaction*. A transaction has the following elements:

- **BEGIN:** Where the transaction strats. In SQL, it is represented by a **BEGIN**.
- **READ or WRITE:** $read(X)$ or $write(X)$ operations. $read(X)$ can also be represented as $R(X)$, while $write(X)$ as $W(X)$.
- **COMMIT:** A successful end to the transaction that *commits* the updates safely to the database. In SQL, it is represented by a **COMMIT**.

- **ROLLABACK:** An unsuccessful end to the transaction, requiring to *undo* the database changes. In SQL, it is represented by an **ABORT**.

The states of a transaction are shown in Fig. 1. A transaction starts in the *active state* after beginning and while executing $read(X)$ or $write(X)$. When the transaction has reached the end, it goes to the *partially committed* state. If there are no errors, the transaction ends as a *committed* transaction. If there is a failure while executing operations or committing the results, the transaction enters a *fail state*. Thus the results to the database are then *aborted*. After being aborted or commited, the transaction is *terminated*.

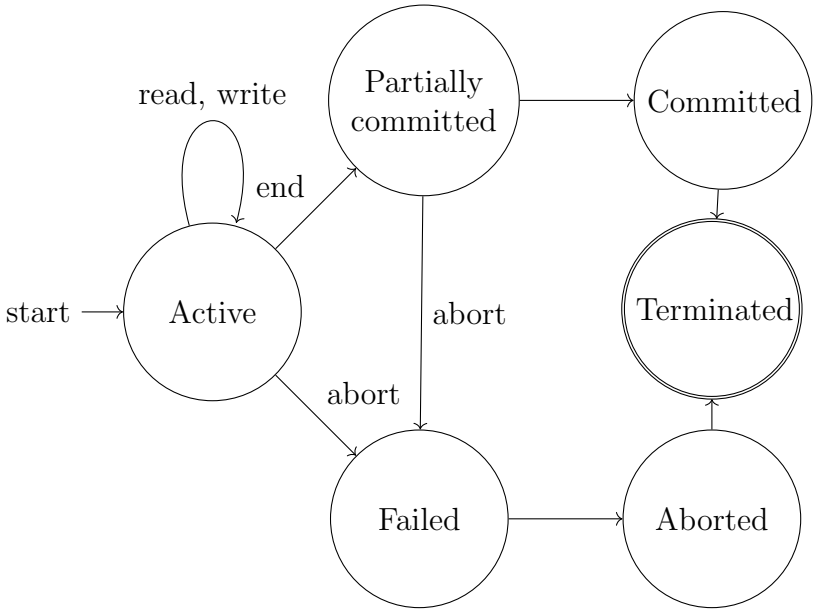

Figure 1: States of a transaction

An example of a transaction T_1 is shown in Fig. 2. In this transaction, \$100 are transfered from Alice's bank account (A) to Bob's bank account (B). This transaction can also be represented in one line as $R(A), W(A), R(B), W(B)$. Another example is T_2 shown in Fig. 3. In T_2, \$50 have been deposited to Alice's bank account (A). Sometimes the **BEGIN** and **COMMIT** statements are omitted, though they do exist for a transaction.

T_1 :
BEGIN;
read(A);
$A := A - 100$;
write(A);
read(B);
$B := B + 100$;
write(B);
COMMIT;

Figure 2: Transfering \$100 from Alice's bank account (A) to Bob's bank account (B)

3 Transaction properties

While transactions are executing, we require that *ACID properties* are maintained for the database. We will describe each property with transaction T_1.

T_2 :
BEGIN;
read(A);
$A := A + 50$;
write(A);
COMMIT;

Figure 3: Cash deposit of \$50 to Alice's bank account (A)

Atomicity

Let us assume that account A has \$1000 and account B has \$500 before the transaction. Suppose that there is a failure after the $write(A)$ operation and before the $write(B)$ operation. Thus, the value would have \$900 for A and \$500 for B, destroying \$100. The database would be in an *inconsistent state* as the system would no longer reflect the real world state. This inconsistent state will happen at one point in the transaction, which will then become consistent after the $write(B)$ operation. Thus, if the transaction was never started or was guaranteed to be completed, the inconsistent state would not be visible except during the transaction.

The *atomicity* means that all operations are executed or none at all. During the transaction the system keeps track of the changes in a *log* file. If there is any failure, the system restores the old values with the log files as if the transaction was not executed. The database system is responsible for ensuring the atomicity through the *recovery system*.

Consistency

We want that the result $A + B$ to be the same *before and after executing the transaction*. Without consistency, money could be destroyed or created (no bueno).

The *consistency* property ensures that executing a transaction by itself preserves the consistency of the database. The *programmer* who codes the transaction is responsible to ensure the consistency.

Isolation

If several transactions are executed concurrently, operations may interleave in such a way that may produce an inconsistent state. If while transferring money from A to B between $write(A)$ and $write(B)$ operation another second transaction read data reads A or B there will be an inconsistent state. If this second transaction updates A and B based on the inconsistent values, the database will be inconsistent after the execution of the transactions.

The *isolation* property ensures that executing the transaction concurrently results in an equivalent state as if it was done one at a time in a particular order. The database ensures this property with the *concurrency-control system*.

Durability

If we have transferred the funds successfully, even if there is a system failure later the data cannot be lost. For example, the system cannot lose that A has \$900 in the account and B has \$600 in the account.

The *durability* property guarantees after successfully completing a transaction, all the updates persist even if there is a system failure. To do this, we must write to disk before finishing the transaction

or by saving enough information to disk to be able to reconstruct the updates after failure. The *recovery system* is also in charge of ensuring durability.

References

[1] R. Elmasri and S. Navathe, *Fundamentals of database systems*, 7th ed. Pearson, 2016, chapter 20.

[2] A. Crotty and M. Li. Lecture #15. [Online]. Available: https://15445.courses.cs.cmu.edu/fall2021/schedule.html

[3] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY: McGraw-Hill, 2020, chapter 17.

