Transaction Systems

Ilaria Battiston *

Summer Semester 2020-2021

*All notes are collected with the aid of material provided by T. Neumann. All images have been retrieved by slides present on the TUM Course Webpage.

Contents

1 Introduction

Transaction systems are essential for real-life applications:

- OLTP systems (online transaction processing), which consist in an architecture with a DBMS and operations happening inside it;
- E-commerce, more complicated since there are external systems in addition to a database;
- Workflows such as travel planning and booking, usually taking more time than the other two.

This kind of scenarios can happen within different modalities: embedded SQL, dynamic SQL and so on. Such techniques allow communication between programming languages and databases in a black-box environment, making the database unaware of the complexity and the meaning of statements.

The main problem is that in the same database, multiple operations can happen at the same time, generating concurrent executions: in the case of reads and writes on the same data, this can generate inconsistencies.

For instance, a write happening immediately after another write can overwrite the previous one, losing information: this scenario needs to be prevented, requiring concurrency control for isolation of transactions.

Furthermore, the system can crash between statements, which is a risk happening regardless of the technical infrastructure (power outages and such) so cannot be avoided: a transaction may not be committed while the next one will, creating potentially inaccurate situations.

Failures therefore need recovery mechanisms for atomicity and durability of data.

Complex systems such as online shopping involve multiple parts (and multiple databases) constantly interacting between one another: distributed systems must employ transactional effects to obtain persistent information, especially since some operations cannot be rolled back.

Workflows are commonly-used computerized parts of business processes, consisting in a set of activities with specific flow and control between them.

These can be represented as Petri networks, and involve an if-else strucure which might spawn other transactions in other systems, gathering information over a long span of time (i. e. booking a conference and finding flights).

Once again, state must be failure-resistant, allowing to save existing operations without isolating its parts if they need to interact.

To split the workload and differentiate logic, usually more layers are used, creating a 3-tier system architecture:

1. Client, with GUIs and websites to be rendered on browsers;
2. Application server, composed by application programs and request brokering (middlewares);
3. Data server, commonly a database, but also a mail server or documents.

Some specific cases only use two tiers, such as a simple client-server in which the application part resides on the client (or on the database).

All the properties mentioned before within requirements of transaction systems can be summarized in ACID:

- Atomicity, either an operation is completely done or rolled back (a rollback must always succeed, even at the cost of shutting down the connection);
- Consistency, transactions must be aborted if properties of the state are violated;
- Isolation, data should be visible as if single-user mode;
- Durability, persistence of storage even during failures.

Every transaction has to follow a begin-commit-rollback flow. If a commit succeeds, it is never lost and it has to be propagated within all the system.

These properties can be achieved by implementing the following components:

1. Concurrency control, to guarantee isolation;
2. Recovery, for atomicity and durability;
3. Performance, trying to maximize throughput to minimize response time;
4. Reliability, never losing data;
5. Availability, providing a continuous server with very short downtime.

Another thing to keep in mind is how to store data, i. e. which storage structure to use: databases employ pages, containing a header and (ideally) continuous data. When an update triggers an overflow, the tuple is placed on a new page and a pointer is added.

Complementary structures such as indexes allow to access information more efficiently, implementing lookup in logarithmic time for both point and range queries (B+-trees). These should also taken into account when implementing concurrency and query plans.

2 Computational models

Computational models in transaction systems work with the assumption of database system layers, in which requests are communicated at the top level of the stack and propagated below; the system must make sure that results are correct, even in an environment with multiple users and operations.

To get this, some elements are needed:

- Elementary operations, primitive;
- Transactions, logical compounds of execution;
- Schedule and history;
- Correctness, expressed by a common notion;
- Protocols, algorithms to enforce the latter.

2.1 Page model

A transactin t is a partial order of seps of the form $r(x)$ or $w(x)$ with $x \in D$, in which reads and writes to the same object are ordered.

$t = (op, <)$ represents a partial order $<$ for transaction t with step set op. This model is quite simple since it only consists in reads and writes defined in order:

1. If $p_j = r(x)$, the interpretation is an assignment of type $v_j = x$;
2. If $p_j = w(x)$, the assignment is performed without knowing the series of previous operations, hence $x = f_j(v_{j1}, \ldots, v_{jk})$ with unknown function.

2.2 Object model

An object model identified a transaction as a tree of labeled nodes, having the following characteristics:

- The identifier corresponds to the root;
- Names and parameters are labels of inner nodes;
- Read/write operations are leaves;
- Leaves are subject of a partial order $<$ such that for $p = w(x)$ and $q = r(x)$ or $q = r(x)$, then $p < q \vee q < p$.

Inner nodes can be ordered by checking their leaf descendants and comparing them.

It is also possible to build trees according to application logic, implementing custom object modules taking advantage of index structures to allow more degrees of freedom.

3 Concurrency control

3.1 Synchronization problems

The most common and problematic synchronization issue is the so-called lost update: a process A updates a value, while process B also performs the update with a different value. The first assignment is lost, since the second transaction gets written over it.

The problem here is an interleaving of reads and writes, of the kind $r_1(x)r_2(x)w_1(x)w_2(x)$: simply swapping the middle ones would solve concurrency issues.

Furthermore, reads can happen inconsistently: in this case, operations can be executed sequentially, yet consequent reads of the same process may lead to wrong results.

Dirty reads are a particular scenario in which a write fails and it is rolled back, invalidating previously read data. Furthermore, successive writes can also be compromised by not knowing the state of the system.

3.2 Schedules

Let T be a set of transactions, in which $t_i = (op_i, <_i)\ \forall i$. Then:

- A history is a pair $s = (op(s), <_s)$ such that:
  - $op(s) \subseteq \cup_{i=1\ldots n} op_i \cup \cup_{i=1\ldots n} \{a_i, c_i\}$;
  - $c_i \in op(s) \leftrightarrow a_i \notin op(s)$;
  - $p <_s c_i \vee q <_s p$ (all operations are ordered before a commit);
  - If at least one p, q is a write and both access the same element, then $p <_s q \vee q <_s p$.

- A schedule is a prefix of a history, allows to run transactions.

A history (partial order) is serial if for any two transactions i, j in s, all operations in t_i are ordered in s strictly before all operations in t_j, or vice versa.

The definition of correctness for a history may vary, however in these examples it is stated as an equivalence relation which must be decidable and sufficiently large. For simplicity, it is assumed that all transactions commit.

3.3 Herbrand semantics

Herbrand semantics are sets of rules defining dependency between writes and consequent reads, since the latter's result is related to the previous operation.

Since functions are deterministic, the same input within the same set of operation must always lead to the same output; therefore, Herbrand semantics can be treated as functions.

A Herbrand universe is the smallest set of symbols such that:

- $F_{0x}() \in HU$ for each $x \in D$ where f_{0x} is a constant;
- If $w_i \in op$ for some t_i, there are m read operations that precede $w_i(x)$ in t_i, (...).

3.4 Serializability

Two schedules are final state equivalent if their Herbrand semantics are the same, and $op(s) = op(s')$: not only the same operations are performed, but also for the same reasons having the same values.

To give more meaning to Herbrand semantics, reads-from relations are introduced by extending a schedule with an initial and final transaction, respectively t_0 and t_∞:

- $r_j(x)$ reads x in s from $w_i(x)$ if $w_i(x)$ is the last write on x such that $w_i(x) <_s t_i(x)$;
- The reads-from relation is $RF(s) = \{(t_i, x, t_j) \mid r_j \text{ reads } x \text{ from a } w_i(x)\}$;
- Step p is directly useful for step q (...)
- Step p is alive if it is useful for the final transaction, and dead otherwise;
- The live-reads-from relation is $LRF(s) = \{(t_i, x, t_j) \mid \text{an alive } r_j \text{ reads } x \text{ from } w_i(x)\}$.