Integrity in Distributed Databases

Andreas Farella

Free University of Bozen-Bolzano
## Table of Contents

1 Introduction .......................................................... 3  
2 Different aspects of integrity ..................................... 3  
  2.1 Concurrency control ............................................. 3  
  2.2 Recovery ............................................................. 4  
  2.3 Application constraints ......................................... 4  
  2.4 Heterogeneity ...................................................... 4  
3 Challenges .............................................................. 4  
  3.1 Example ............................................................... 4  
  3.2 Integrity issues ..................................................... 6  
4 Proposed solution: transaction management ..................... 6  
5 Integrity applied ........................................................ 7  
  5.1 CAP Theorem ....................................................... 7  
  5.2 F1 - Google Database System ................................. 9  
  5.3 F1 - Basic architecture ......................................... 9  
  5.4 F1 - Paxos algorithm ............................................ 10  
6 Conclusion .............................................................. 11
1 Introduction

As the data handled by organizations grows bigger every year, digitalization of such data into databases is a must in order to work effectively. As many heterogeneous databases are scattered across several sites they need to be interconnected in order to allow access to the whole data, not just local partitions. Access to the data not only has to be efficient, but the stored data must be ensured to be as accurate and up-to-date with the real world as possible.

Maintaining integrity while allowing concurrent access is a real challenge which has to be faced in a distributed environment of (possible) heterogeneous database systems. In this paper various aspects and possible solutions are discussed and then compared to how it is handled in reality, by using Google’s F1 database system as example.

2 Different aspects of integrity

Integrity in a traditional database systems is mostly described through the ACID (Atomicity, Consistency, Isolation, Durability) properties:

– Atomicity:
  Either all instructions of a transaction are executed or none.
– Consistency:
  A transaction must transform the database from one consistent state to another.
– Isolation:
  Transactions do not access data being modified by other transactions.
– Durability:
  Committed data is stored permanently to the database.

Besides these ACID properties in a distributed environment further aspects need to be considered, e.g. one big issue is to handle partitions in case of network/nodes failure.

2.1 Concurrency control

In a local database system concurrent access to the data is handled by the DBMS itself. In a distributed environment some transactions may access multiple sites at once, generating sub-transaction for each site which is accessed. These sub-transactions are usually executed concurrently in order to have a good overall performance.

Further, different copies of the same data may be scattered across the system. When a transaction executes, it has to be ensured that all replications of the affected data remain consistent.
2.2 Recovery

Some transactions in a distributed database system are executed at different sites, which are connected through the network which is prone to various failures. Because of this, whenever a sub-transaction has already been executed but the whole transaction is ultimately aborted, the data has to be rolled back to the last consistent state.

Also, some sites may be fail to be accessed, in which case a copy of the data is accessed. When the site connects back to the system, it has to be updated again to the actual data.

2.3 Application constraints

The applications running on top of the database may have need for special constrains, which in turn must be enforced across multiple databases. When multiple application run concurrently, it has to be ensured that the constrains are not conflicting or violated across each other.

2.4 Heterogeneity

The distinct databases inside the network may differ significantly from each other because of the system they use. In order for the data to be used correctly inside the distributed environment, it has to be ensured that the semantics and constrains of the distinct systems are maintained.

3 Challenges

3.1 Example

A distributed DBMS can be defined in the following way:

A distributed database is a collection of data which is distributed over different computers of a computer network. Each site of the network has autonomous processing capability and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication subsystem. A distributed database management system supports the creation and maintenance of a distributed database.
As a concrete example of such a system, let's consider the University of Bozen-Bolzano. As it is known, the main institution is located in Bozen, however there are also two smaller branches in Brixen and Bruneck. Each of these branches offers different majors, and thus has distinct employees and departments. Let's have a simplified look at it:

<table>
<thead>
<tr>
<th>StudentID</th>
<th>Name</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dallas</td>
<td>Computer Science</td>
</tr>
<tr>
<td>2</td>
<td>Hoxton</td>
<td>Economics</td>
</tr>
<tr>
<td>3</td>
<td>Chains</td>
<td>Design</td>
</tr>
<tr>
<td>4</td>
<td>Wolf</td>
<td>Economics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CourseID</th>
<th>Name</th>
<th>Major</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Introduction to Programming</td>
<td>Computer Science</td>
<td>Bozen</td>
</tr>
<tr>
<td>20</td>
<td>Economics 101</td>
<td>Economics</td>
<td>Bozen</td>
</tr>
<tr>
<td>30</td>
<td>Education I</td>
<td>Education</td>
<td>Brixen</td>
</tr>
<tr>
<td>40</td>
<td>Outdoor Photography</td>
<td>Design</td>
<td>Bozen</td>
</tr>
</tbody>
</table>
3.2 Integrity issues

Already with the previous small example there are some issues which have to be considered:

– Referential integrity:
  The relevant tables may be scattered across multiple sites. When inserting new tuple in a table, a remote site may has to be queried in order to ensure the referential integrity (e.g. foreign keys constrains). Opposite to that, the tables may be replicated across multiple sites, each holding a portion of the values (horizontally fragmented). Also in this case a remote site may has to be queried (e.g. primary key constrains).

– Application specific integrity:
  The applications running on the different sites may enforce themselves some integrity on the data, e.g. the total salary of all employees may not exceed 20 million, an employee can manage at most one department. There are many possible ways to handle such constrains: should they be distributed across the sites? Should the all be centralized in one site? Should the be replicated across all sites?

– Heterogeneity integrity:
  The different sites may not be equal to each other, that is they may differ in the employed data model or use distinct DBMS. To maintain integrity across all sites, some special considerations need to be taken.

4 Proposed solution: transaction management

When accessing a local DBMS the transaction management is handled by the system itself, however it is not so straight forward in a distributed environment. Usually there are distributed transactions, each performing a sub-transaction at a given site.

Executing transactions serially is not an option as this would cause a serious performance drawback. Executing multiple transactions in parallel should result in the database being in a consistent state. Further attention is needed when data is duplicated or certain sites fail.

In order to ensure consistency, a transaction should be serializable. A sequence of operations performed by multiple transactions is called a schedule. If the result of two different schedules is the same, then the schedules are equivalent to each other. When inside a schedule no two transactions execute in parallel, then the schedule is said to be a serial schedule. The aim in a DDBMS is to ensure that any schedule is a serializable schedule, which means to ensure that any schedule is equivalent to a serial schedule (the transactions, schedules and their operations can be executed in a serial fashion). To ensure serialization of schedules there are various techniques like locking, time-stamping and validation.

– Locking:
  To ensure concurrency control two-phase locking may be used: first all the
required locks are obtained before any are released. For a read operation usually a shared lock is requested, which is granted whenever there is no write lock on the same tuple. For a write operation an exclusive lock is normally requested, which is granted when there is no lock on the tuple. When having multiple copies of the same data item, some further considerations have to be made:

- Shared locks are acquired over any copy of the data, exclusive locks are obtained over all the copies of the data.
- Locks are obtained over the majority of the copies.
- Locks are requested only on the primary copy of the data.

- Time-stamping:
  Each transaction has a time-stamp of when it begun. Each tuple has assigned a read and write stamp, representing the most recent transaction which accessed the tuple for read or write. When a read or write is requested, the time-stamps are compared with each other. If the checks succeed the transaction is execute, else it is restarted with a new time-stamp.
  In a DDBMS usually a global time-stamp is used, in order to synchronize the system clocks at the different sites.

- Validation (optimistic concurrency control technique):
  Before being committed to the database, each sub-transaction is validated first. If it fails it gets aborted, and in turn the whole transaction is aborted.
  If the local validation succeeds, a global validation is started. If it fails the global validation it is aborted again, whereas if all sub-transactions pass the global validation the whole transaction is committed.

A popular technique to avoid inconsistencies due to failed or aborted transactions is the two-phase commit protocol. The site where the transaction is started usually acts as coordinator and the sites where the sub-transactions are executed act as participants. Both maintain a log of their operations. When committing a transaction, the coordinator send a message to all its participants to prepare for commit and each participant in turn replies with whether it is ready to commit. If all the participants answer positively, then the coordinator instructs the participants to commit, else the participants are instructed to abort the operation. According to the received message, the participant either commits or aborts the sub-transaction.

5 Integrity applied

A few issues and possible solutions regarding integrity in DDBMS as they are found in literature have been discussed. Unluckily, literature and reality often do not coincide. The focus on the next part is to see how integrity is dealt with in real situations.

5.1 CAP Theorem

It is proven that it is possible to achieve the ACID properties in a locally closed system, however it seems to be not possible to do so in a distributed environment,
where some qualities have to be sacrificed for others. This dilemma is known as the CAP Theorem:

- **Consistency:**
  All nodes see the same data at the same time.

- **Availability:**
  Every request is expected to receive an answer to whether it succeeded or failed, which means there should always exist a node which can be reached.

- **Partition tolerance:**
  The system can still operate properly whenever messages are lost due to sudden network partitioning, or parts of the system fail or are isolated.

The CAP Theorem states, that it is infeasible for a distributed DBMS to achieve all three of these properties, as one can never be satisfied (“two out of three”), which still makes it stronger compared to the ACID properties which are also to be considered. However, there is no unanimity among the experts: according to some this is to strict and they want to move even further away from the ACID properties, others instead strife for ACID as they claim it can indeed be achieved.
5.2 F1 - Google Database System

Given the high amount of data dealt with by Google, their worldwide partition and the large amount of incoming queries, they have to take integrity in their distributed environment really seriously. Because of this, they had their own in-house build DBMS, called F1. They claim to have high availability, scalability like NoSQL systems (e.g. BigTable) and consistency and usability of traditional SQL databases, at the cost of high commit latency (however, they claim to mitigate that by using a hierarchical schema model with structured data types and smart application design). Prior to F1, they had a sharded MySQL implementation, which couldn’t keep up with their needs for scalability and reliability.

F1 is based on spanners, which is Google’s distributed NewSQL database, which again is the successor to Google’s BigTable storage system. It makes explicit data clustering by using tables with hierarchical relationships and columns with structured data types, which results in improved data locality and reduced number and cost of remote procedure calls required to read data on a remote node. This is further improved by making heavy use of batching, parallelism, asynchronous reads and a new object-relational-mapping (ORM), which apparently places an upper bound to the number of RPC required to perform an operations, improving their scalability.

5.3 F1 - Basic architecture

Users interact with F1 through a F1 client, which forwards the requests to a F1 server, which is responsible to access local and remote data sources and coordinate execution.
Load Balancer:
The load balancer redirects the requests to an appropriate/available server, in order to distribute the load among the servers. Whenever possible, requests are redirected to a server which is close to which the request originated.

F1 Server
Inside a data center there are multiple F1 servers, in order to increase availability and fault tolerance in case of failure of some servers. The servers are usually physically located at the same place where the spanners are, to increase fast access to the underlying data, but they can also access remote spanners. The server is stateless.

F1 Master - Slave Pool
Internally, the query are executed by the available slaves, which act as F1 processes. The F1 master’s role is to overwatch the progress of the slaves and distribute work among them, in a Map Reduce fashion. To increase performance, the slaves are allowed to directly access the spanners.

Spanner
Spanners are Google’s NewSQL database system and they retrieve the data from the Colossus File System on the same data center. They mainly handle low-level storage like caching and replication. Internally the data is partitioned into directories. To access the data, a strict two-phase-commit protocol with timestamping is used.

Colossus File System (CFS)
The Colossus File System is the distributed file system by Google which runs on commodity hardware and aims at reliable access over large clusters.

Whenever needed, additional servers and slaves can be added to the system seamlessly, as no movement of data is involved in the process. GPS clocks and atomic clocks are employed to ensure global consistency.

5.4 F1 - Paxos algorithm

Regarding synchronization, Google uses the Paxos algorithm to ensure consensus over unreliable processors. To propose a change a majority of the involved nodes must approve it, which is somewhat loosely related to the two phase commit protocol, which also uses multiple participants and their response.

Paxos algorithm is an algorithm used to agree on a value among its participants inside an unreliable network by using quorum consensus (the majority of the participants has to agree on the value). Whenever a new value has to be agreed on, the following steps are performed:

- Prepare
  A proposer prepares a proposal with number N which is send to a quorum of acceptors. The proposer decides who is in the quorum.

- Promise
  If the number N of the proposal is higher than any previous proposal number
received from any proposer by the acceptor, then the acceptor must return a promise to ignore all future proposals having a number less than N. If the acceptor already accepted a proposal at some point in the past, it must include the previous proposal number N and previous value in its response.

– Accept request
If the proposer receives enough promises from the participants it can decide upon a value. If no participant has replied with a value of a past proposal a new value can be proposed. Otherwise, the value of the proposal with the highest number is proposed. The value is then sent to the quorum of acceptors as an accept request.

– Accepted
If the participants receive an accept request they will accept its value only if they have not already promised on a proposal. Upon accepting a value an accepted message is send back.

Inside the spanner directory fragments are stored into groups, which usually has a so-called replica tablet per data center. All tables belonging to a group store the same data and one of them is elected as the Paxos leader for that group, which functions as an entry point for transactional activity for that group. Read-only replicas do not take part in the vote and can also not become leader of a group.

Whenever a commit occurs, it gets synchronously replicated via Paxos. Updating data co-located in a single group is fastest, for updates across multiple groups two phase commit protocol on top of Paxos is used.

6 Conclusion
Some established techniques have been discussed and comparison with a real-life situation with Google’s F1 database have been made. Tough some techniques appear to be basic, F1 proves that they are still valid and can be employed successfully with some adaptations. Overall it is clear, that integrity in distributed DBMS is feasible, tough not straight forward.

References