Highly Available Database Management Systems

Wenbing Zhao
Cleveland State University, USA

INTRODUCTION

In the Internet age, real-time Web-based services are becoming more pervasive every day. They span virtually all business and government sectors, and typically have a large number of users. Many such services require continuous operation, 24 hours a day, seven days a week. Any extended disruption in services, including both planned and unplanned downtime, can result in significant financial loss and negative social effects. Consequently, the systems providing these services must be made highly available.

A Web-based service is typically powered by a multi-tier system, consisting of Web servers, application servers, and database management systems, running in a server farm environment. The Web servers handle direct Web traffic and pass requests that need further processing to the application servers. The application servers process the requests according to the predefined business logic. The database management systems store and manage all mission-critical data and application states so that the Web servers and application servers can be programmed as stateless servers. (Some application servers may cache information, or keep session state. However, the loss of such state may reduce performance temporarily or may be slightly annoying to the affected user, but not critical.) This design is driven by the demand for high scalability (to support a large number of users) and high availability (to provide services all the time). If the number of users has increased, more Web servers and application servers can be added dynamically. If a Web server or an application server fails, the next request can be routed to another server for processing.

Inevitably, this design increases the burden and importance of the database management systems. However, this is not done without good reason. Web applications often need to access and generate a huge amount of data on requests from a large number of users. A database management system can store and manage the data in a well-organized and structured way (often using the relational model). It also provides highly efficient concurrency control on accesses to shared data.

While it is relatively straightforward to ensure high availability for Web servers and application servers by simply running multiple copies in the stateless design, it is not so for a database management system, which in general has abundant state. The subject of highly available database systems has been studied for more than two decades, and there exist many alternative solutions (Agrawal, El Abbadi, & Steinke, 1997; Kemme, & Alonso, 2000; Patino-Martinez, Jimenez-Peris, Kemme, & Alonso, 2005). In this article, we provide an overview of two of the most popular database high availability strategies, namely database replication and database clustering. The emphasis is given to those that have been adopted and implemented by major database management systems (Davies & Fisk, 2006; Ault & Tumma, 2003).

BACKGROUND

A database management system consists of a set of data and a number of processes that manage the data. These processes are often collectively referred to as database servers. The core programming model used in database management systems is called transaction processing. In this programming model, a group of read and write operations on a data set are demarcated within a transaction. A transaction has the following ACID properties (Gray & Reuter, 1993):

- **Atomicity:** All operations on the data set agree on the same outcome. Either all the operations succeed (the transaction commits) or none of them do (the transaction aborts).
- **Consistency:** If the database is consistent at the beginning of a transaction, then the database remains consistent after the transaction commits.
- **Isolation:** A transaction does not read or overwrite a data item that has been accessed by another concurrent transaction.
- **Durability:** The update to the data set becomes permanent once the transaction is committed.

To support multiple concurrent users, a database management system uses sophisticated concurrency control algorithms to ensure the isolation of different transactions even if they access some shared data concurrently (Bernstein, Hadzilacos, & Goodman, 1987). The strongest isolation can be achieved by imposing a serializable order on all conflicting read and write operations of a set of transactions so that the transactions appear to be executed sequentially. Two operations are said to be *conflicting* if both operations access the same data item, at least one of them is a write operation, and they belong to different transactions. Another popular isolation model is snapshot isolation. Under the snapshot

isolation model, a transaction performs its operations against a snapshot of the database taken at the start of the transaction. The transaction will be committed if the write operations do not conflict with any other transaction that has committed since the snapshot was taken. The snapshot isolation model can provide better concurrent execution than the serializable isolation model.

A major challenge in database replication, the basic method to achieve high availability, is that it is not acceptable to reduce the concurrency levels. This is in sharp contrast to the replication requirement in some other field, which often assumes that the replicas are single-threaded and deterministic (Castro & Liskov, 2002).

DATABASE HIGH AVAILABILITY TECHNIQUES

To achieve high availability, a database system must try to maximize the time to operate correctly without a fault and minimize the time to recover from a fault. The transaction processing model used in database management systems has some degree of fault tolerance in that a fault normally cannot corrupt the integrity of the database. If a fault occurs, all ongoing transactions will be aborted on recovery. However, the recovery time would be too long to satisfy the high availability requirement. To effectively minimize the recovery time, redundant hardware and software must be used. Many types of hardware fault can in fact be masked. For example, power failures can be masked by using redundant power supplies, and local communication system failures can be masked by using redundant network interface cards, cables, and switches. Storage medium failures can be masked by using RAID (redundant array of inexpensive disks) or similar techniques.

To tolerate the failures of database servers, several server instances (instead of one) must be used so that if one fails, another instance can take over. The most common techniques are database replication and database clustering. These two techniques are not completely distinct from each other, however. Database replication is typically used to protect against total site failures. In database replication, two or more redundant database systems operate in different sites — ideally in different geographical regions — and communicate with each other using messages over a (possibly redundant) communication channel. Database clustering is used to provide high availability for a local site. There are two competing approaches in database clustering. One uses a shared-everything (also referred to as shared-disk) design, such as the Oracle Real Application Cluster (RAC) (Ault & Tumma, 2003). The other follows a shared-nothing strategy, such as the MySQL Cluster (Davies & Fisk, 2006) and most DB2 shared database systems. To achieve maximum fault tolerance and hence high availability, one can combine database replication with database clustering.

Database Replication

Database replication means that there are two or more instances of database management systems, including server processes, data files, and logs, running on different sites. Usually one of the replicas is designated as the primary, and the rest of the replicas are backups. The primary accepts users' requests and propagates the changes to the database to the backups. In some systems, the backups are allowed to accept read-only queries. It is also possible to configure all replicas to handle users' requests directly. But doing so increases the complexity of concurrency control and the risk of more frequent transaction aborts.

Depending on how and when changes to the database are propagated across the replicas, there are two different database replication styles, often referred to as *eager* replication and *lazy* replication (Gray & Reuter, 1993). In eager replication, the changes (i.e., the redo log) are transferred to the backups synchronously before the commit of a transaction. In lazy replication, the changes are transferred asynchronously from the primary to the backups after the transactions have been committed. Because of the high communication cost, eager replication is rarely used to protect site failures where the primary and the backups are usually far apart. (Eager replication has been used in some shared-nothing database clusters.)

Eager Replication

To ensure strong replica consistency, the primary must propagate the changes to the backups within the boundary of a transaction. For this, a distributed commit protocol is needed to coordinate the commitment of each transaction across all replicas. The benefit for doing eager replication is that if the primary fails, a backup can take over instantly as soon as it detects the primary failure.

The most popular distributed commit protocol is the two-phase commit (2PC) protocol (Gray & Reuter, 1993). The 2PC protocol guarantees the atomicity of a transaction across all replicas in two phases. In the first phase, the primary (which serves as the coordinator for the protocol) sends a *prepare* request to all backups. If a backup can successfully log the changes, so that it can perform the update even in the presence of a fault, it responds with a "Yes" vote. If the primary collects "Yes" votes from all backups, it decides to *commit* the transaction. If it receives even a single "No" vote or it times out a backup, the primary decides to *abort* the transaction. In the second phase, the primary *notifies* the backups of its decision. Each backup then either commits or aborts the transaction locally according to the primary's decision and sends an acknowledgment to the primary.

As can be seen, the 2PC protocol incurs significant communication overhead. There are also other problems such as the potential blocking if the primary fails after all backups