IBExpert White Paper

Why the order of write operations is so important for Firebird

Holger Klemt, September 2026

The reliability of a Firebird database does not depend on Firebird alone. It is equally important that the underlying operating system and storage system execute and acknowledge write operations in the way the database expects.

Firebird is designed to write changes to storage in a defined and carefully coordinated order. In particular, when Forced Writes are enabled, Firebird ensures that critical changes have actually been written to persistent storage before subsequent operations are considered successfully completed.

Caches and additional storage layers

Modern operating systems, RAID controllers and storage systems attempt to optimize write operations for performance reasons. For example, write operations may be collected, combined or passed to the physical storage device in a different order.

This is generally useful and usually unproblematic for normal file access. For a database system, however, it is essential that so-called flush or synchronization requests are reliably passed through all layers down to the actual persistent storage medium.

Additional layers between Firebird and the physical storage device can therefore become critical, for example:

- Host or controller caches
- RAID and storage virtualization
- Cluster file systems
- Block-level replication
- SAN or NAS systems
- Snapshots and other virtualization layers
- Write caches that are not protected by battery backup or flash-based protection

The problem arises in particular when one of these layers reports a write operation as completed even though the data has not yet been permanently stored.

What can happen during a crash

In the event of a power failure, kernel crash or hardware failure, write operations that have already been acknowledged may therefore be lost.

This becomes particularly problematic when a later database state has already reached the storage device while related earlier changes are still waiting in a cache. This can result in information within the database file that is no longer consistent.

Firebird's carefully designed write strategy is intended to prevent exactly these situations. However, this requires the operating system, drivers, controllers and storage system to correctly implement the corresponding synchronization requirements.

Disabling Forced Writes may therefore significantly improve write performance, but at the same time it removes an important layer of database protection.

Linux is not automatically protected against these problems

Linux file systems such as ext4 provide mechanisms for controlling the order and durability of write operations. However, this does not automatically mean that every underlying storage configuration provides the same guarantees.

As soon as additional cluster, mirroring, RAID or virtualization layers are involved, it must be ensured that flush operations and write barriers are actually passed through to persistent storage.

For database systems, the more important question is therefore not whether Windows or Linux is being used, but whether the *entire storage stack* reliably provides the guarantees required by the database system.

A real-world example

Several years ago, a serious failure occurred in a business-critical Firebird environment managed by us. The database was hosted on an externally implemented Linux cluster and storage solution.

Following an error, not only the production database but also the copy created at storage level was affected. Restoring the system required extensive analysis and repair work.

The architecture was subsequently changed. Since then, redundancy has no longer been provided through transparent mirroring of an open database file, but through replication at the Firebird database level to independent server systems. This architecture has been operating reliably in that environment for many years.

Redundancy should ideally be implemented at database level

For Firebird systems, we therefore prefer storage architectures that are as simple and transparent as possible.

Where possible, redundancy and high availability should be implemented in such a way that two independently operating Firebird servers each maintain their own local database file, with synchronization being controlled at the database level.

This approach offers several advantages:

- Each Firebird server controls its own database file.
- Storage problems affecting one system are not automatically transferred to the second database file.
- The number of additional caching and virtualization layers is reduced.
- Problems are easier to analyze and isolate.
- Firebird's write mechanisms, which are essential for data integrity, remain intact.

Security and performance are not mutually exclusive

Strictly maintaining the required order of write operations inevitably introduces a certain amount of latency. In applications with very intensive write activity, this latency can be clearly measurable.

However, this additional latency is not unnecessary performance loss. It is an essential component of transaction safety.

When optimizing a Firebird server, the goal should therefore not be to circumvent these safety mechanisms. A better approach is to design the hardware and storage architecture in such a way that synchronous write operations can be completed as quickly as possible.

A fast database requires fast storage. A reliable database, however, requires storage whose claims regarding successfully stored data actually hold true.