STORMATICS

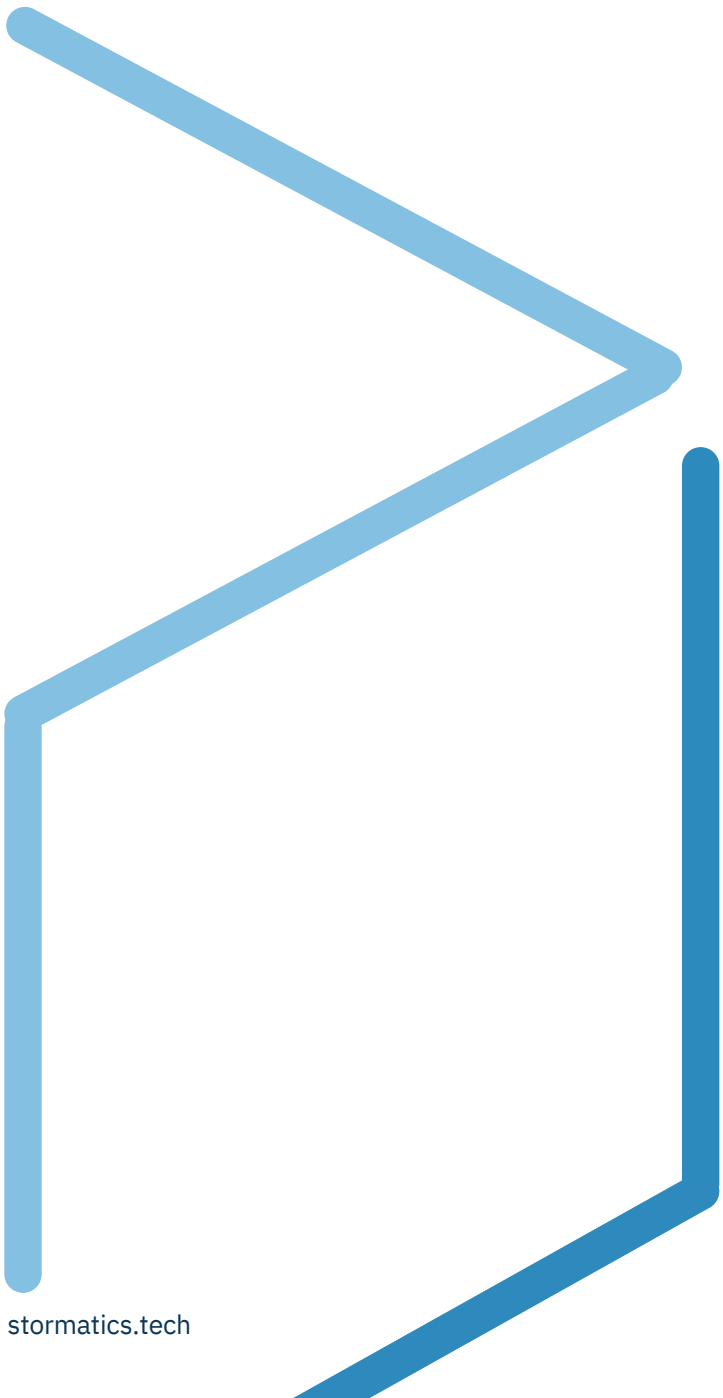
Transparent Data Encryption and High Availability

Contents

Abstract

Transparent Data Encryption (TDE) has become a requirement for many businesses holding sensitive data, whether personal or financial. However, this effect generally means that backups are not directly useful without a separate, stored encryption key, and that starting the server itself might not be done in a fully automated way (depending on how key management is configured). The interaction between these components is not always intuitive. This paper aims to set things straight.

This planning document provides a basic introduction or review to both High Availability and Transparent Data Encryption, and then provides an overview of the operational challenges associated with running these two together. Recommendations focus on both infrastructure to consider and processes to be documented, tested, and rehearsed.



Introducing High Availability

Databases back many applications which the customers or users expect to be "always on." These applications often cannot afford downtime due to maintenance, especially unplanned maintenance. When unexpected issues strike, it is important that the application can continue to keep working.

For stateless applications, this is easy since they do not rely on data they store themselves, and one can just simply install the application onto additional systems. This is not the case for databases, which maintain the shared state of these applications. Additionally, relational databases, such as PostgreSQL are built with the high standards of consistency required for financial applications and therefore making sure not only that the state is shared but that writes are applied in a fully consistent manner are important considerations. These guarantees are often summarized as the "ACID" guarantees.

In PostgreSQL, the primary way in which we ensure both of these is through binary streaming replication. In this model, write requests come into a write master[1], also called a primary, and then are copied to replicas, also called standby systems. The copy deltas are actually binary patches to the on-disk data files, meaning that we can guarantee that when we reach a certain point in the stream of writes, the binary files storing data on a given system will be the same. The replicas then replay these binary patch files against their own files, bringing them up to speed.

We can usually read from these replicas (or not, depending on how they are configured) but while they are replaying changes from the primary, the replicas cannot be written to. When the primary becomes unavailable for any reason, writes stop until the group of systems can be reconfigured to allow the writes to apply against a different database server. This avoids the downtime and potential data loss associated with recovering from backups, but some data loss is still possible if these binary change-files haven't been received when the primary goes down. That data loss is usually a lot less than restoring from a non-continuous-backup however, and we don't need to copy the backup and log segments to another system and have them catch up.

The process of switching which system is handling writes is called "failover" if something has failed, and "switchover" if everything is still in a healthy state when the action occurs. This consists of three steps:

1. Disconnecting and shutting down the previous primary, or write master
2. Determining which standby to use as the new primary, and reconfiguring other standbys to replicate from it.
3. Telling the new primary to stop recovering from these binary logs and accept writes.

[1] The specific terminology here used to be master/slave which is not only culturally insensitive but not very accurate. Master/copy (as used media production) would have been clear. The industry is standardizing on master/replica and primary/replica which are both acceptably clear and avoid inaccurate, insensitive references. However, as some people may be used to this terminology, it's worth noting it here.

After this, the old primary can be told to replicate from the new primary. This is very straightforward if everything is up to date and the switchover is controlled. If the primary had failed, though, it may have to be "rewound" before this can take place. **In all cases, this requires a restart of the old primary.**

High Availability Strategies

There are several important considerations when choosing a high availability solution. These include whether the failover needs to be manual or automatic.

The simplest approach is a manual failover. This can be done either with hand-written scripts, or, more often, with systems capable of doing automatic failover possibly in other conditions. The main reason to use systems that are otherwise capable of automatic failover is that they already are able to automate the steps needed, and this avoids a number of error-prone human steps which are even more error prone when under the stress of an emergency.

Automatic failovers may happen more frequently than strictly needed, and they have a number of strict requirements which manual failovers can get around. One of the primary problems that has to be addressed is the problem of "split brain" where two partitions of a high availability cluster each believe they are responsible for processing writes, and it becomes difficult or impossible to merge the writes in later. The simple solution here is to ensure that a majority of members of a high availability cluster can all see each other (this is called a quorum). For this reason it is important to have an odd number of databases in the high availability set.

If this cannot be done, sometimes a witness node running the failover[2] software, but not the database, can be set up preferably in a neutral third location. Of course the witness node can be set up near one part of the cluster but to do so adds weight on that side and reduces the ability of the other nodes to take over if something goes wrong. For example if you have three nodes split between two datacenters, if the datacenter with two nodes in it is unreachable, then the other node cannot take over.

After an automatic failover, until the old primary is returned to the system as a replica, the conditions no longer allow for perfect confidence in automatic failovers, so manual failover processes must still be considered.

It is important to remember here that all failovers require restarting the database, and this is an important factor to plan for as one goes forward.

EnterpriseDB's Enterprise Failover Manager

EnterpriseDB offers their Enterprise Failover Manager (EFM) project which provides a system that maintains a consistent view of the state of the database systems and is capable of automating the failover and switchover of databases. EFM can be configured to automatically failover if the conditions mentioned above are present, and can be used to orchestrate manual failovers if they are not.

Introducing Transparent Data Encryption

The second component of the topic of this paper is that of Transparent Data Encryption or TDE. At the time of this writing, TDE is only available in proprietary forks of Postgres. There are efforts, however, to include TDE within the core software. TDE provides for cases where a given PostgreSQL system will encrypt all data at rest including both the data files themselves and the write-ahead log segments (which are used both in crash recovery and as the binary patch sets used for binary replication).

The purpose is to protect files and file systems if they are somehow copied off the system by an attacker. This risk includes stolen hard disks but also unauthorized access to the system as the Postgres user. Depending on the configuration of the Linux kernel, however, a root user may be able to read the running PostgreSQL program from memory[2] and attack the key that way. Currently, that is considered an exotic attack vector but it is important to note.

In a system with TDE enabled, PostgreSQL loads up a key when the database starts and uses this key to decrypt data on read and encrypt data on write. It is very important to the security of TDE that the key is not directly accessible to Postgres on system start.

EnterpriseDB offers a general key management framework built into their Postgres Extended and Advanced Server products. This allows the administrator to specify a command to retrieve a working key from the system using a shell command. EDB actively supports key management systems such as Hashicorp Vault as well as password protected keys.

[2] This can be done if access to main memory through /dev/mem is available on the kernel. The presence of /dev/mem does not mean that this attack will work and it depends on compile time options for the kernel. To test, you can: sudo cat /dev/mem > /dev/null and if you get an "operation not permitted" error then you are not vulnerable. To date, most Linux distributions are not believed to be vulnerable as they compile the kernel as locked down against this sort of access.

As one note, most modern CPUs accelerate AES encryption. In these cases the performance impact of TDE is negligible. However if your CPU does not accelerate AES encryption, the performance impact may be significant. If in doubt, please check your hardware documentation.

High Availability and Key Management Problems

If password protected keys are used, then an administrator must enter the password at database startup. This means that the unattended startup of PostgreSQL becomes effectively impossible. In the case of manual failover, this adds important manual steps to the end of the process. In the case of automatic failover, the old primary cannot be started up after shutdown automatically.

Without access to the key, starting Postgres becomes impossible, and since this is never stored in the backups, backups are also useless without the key. This makes key management an absolutely vital part of data assurance.

The use of key management systems alleviates this problem, but they provide another single point of failure when it comes to starting up the database systems. In the case of Hashicorp Vault, high availability is part of additional paid offerings.

In the end, the key management provisions of any TDE solution add significant operational headaches to any database deployment that require careful planning,

Recommendations

Given the specific difficulties that come with the protections TDE, the following recommendations should apply to the planning stage.

Test for /dev/mem Access

Most Linux distributions today, particularly when running on secure boot devices, lock down access to main memory through /dev/mem. This should be periodically tested to make sure that a kernel is not shipped which provides access to main memory (from which the working encryption key could be read). To test this:

```
$ sudo cat /dev/mem > /dev/null
Operation not permitted
```

Here the operation not permitted error states that the kernel is not permitting access to main memory through the /dev/mem device. Typically only one MB will be returned for a locked down kernel and this is considered safe.

In the event where the command completes successfully, your system can have the Postgres process and the encryption key read from memory by anyone with read access to the /dev/mem file. Usually, this is restricted to root and the kmem group, but it is not an acceptable risk for systems that need TDE.

Plan for, Test, and Document Manual and Automatic Failover Processes

As noted above, even when automatic failover is desired, there is a window in which decisions considering manual failover may be required. EFM does not automatically rewind old primaries, and so preparation for manual failover needs to be a part of the process.

This documentation should be kept as part of a runbook or operating manual, and operations personnel should periodically run through it to make sure they understand the procedure. During stressful circumstances, we all follow what we are used to so it is good to be used to these processes.

Thoroughly Understand the Limitations of Backups of the Key Management System

Key management systems have many limitations in their backup routines. Many may refuse to backup certain secrets, and you don't want to find this out during an emergency. If the key is not available, neither the existing Postgres systems nor the binary backups will be of any use.

For this reason, the key management system should be seen as a system of convenience and additional measures should be taken to ensure key availability beyond it.

Consider Paid Options for Key Management Systems

The key management system is an important part of the overall system here. As such when it is down, the database system may not be immediately affected, but it cannot be started once it is shut down. Ensuring proper availability therefore can be an important consideration.

In the case of Hashicorp Vault, High Availability is part of their commercial offering and so in the event that one has the hardware and resources available for a HA solution there, the commercial offerings make a lot of sense.

Back Up the Encryption Key and Password-Protect It

In addition to ensuring availability of the key management system, you should also back up the encryption keys in a password-protected format. This allows a number of options that may not be obvious at first glance:

1. This allows you to restore a backup to a temporary location which cannot access the key management system, for example to fix data from a bad SQL query.
2. This allows you to start up the system in an emergency without the key management system being available.

Both of these scenarios can make the difference between data loss and data availability.

Document and Test Procedures for Starting the Database Without Key Management System

In order to be able to start the database with the password-protected key, it is important to know how to do this. In emergency situations we perform what we have trained, and we tend to lose an ability to think holistically about problems. In the event where the system is not available, and the key management system is down, you want to know how to address it. Having documentation which has been tested and rehearsed is critical to relieving stress and providing for positive outcomes in these cases.

Why Choose Stormatics?

Our global consultants have wide-ranging expertise in running PostgreSQL in the cloud, data centers, on-premise, and hybrid environments.

- **Reliable Solutions**
  You want your database to be reliable, we have the expertise to make sure you can depend on PostgreSQL at scale.
- **Customer Satisfaction**
  Our customers are our biggest asset, and our focus is on excellence in service delivery. We will not rest till you are 100% satisfied.
- **Customized Services**
  Your challenges are unique, and so are our services. Our team collaborates with yours to deliver customized solutions.

We are your trusted PostgreSQL experts

Book a call with us today