PostgreSQL'de Point-In-Time-Recovery ve Warm Standby Database Kavramları

Son Güncelleme: 27 Aralık 2007 Perşembe, 15:40

Giriş

PostgreSQL'in 7.1 sürümünde gelen WAL kavramı, PostgreSQL'in veri kurtarma işleminde tek yönlü olarak (REDO işlemi) oldukça faydalı idi. PostgreSQL 8.0 ile gelen PITR (Point In Time Recovery), veri kurtarma ve yedeklemede bir adım daha ileti gitmiş ve UNDO işleminin de yapılabilir olmasını sağlamıştır. PostgreSQL 8.2 sürümü ile gelen Warm Standby özelliği de PITR yeteneklerini daha ileriye taşımıştır.

Bu yazıda PostgreSQL'de WAL kavramı, UNDO ve REDO işlemleri, PITR teknik özellikleri, PITR yapma süreci ve Warm Standby özellikleri anlatılmıştır. Warm Standby özelliği nedeniyle yazı genel olarak 8.2 ekseninde gidecektir; ancak yönergeler büyük çoğunlukla 8.0 ve 8.1 sürümleri için de kullanılabilir.

PITR işleminin son kullanıcılar için olmadığını ve önemli sayılabilecek miktarda PostgreSQL ve işletim sistemi bilgisi gerektiğini belirtmek isterim. PITR küçük veritabanları için değildir – küçük veritabanları için pg_dump ile yedek alabilirsiniz.

PostgreSQL'de WAL kavramı

WAL (Write Ahead Logging), PostgreSQL'de transactionların kaydedilmesi işlemidir. WAL ile ilgili bazı bilgilere aşağıda ulaşabilirsiniz. Yazıda transaction loglara `xlog` da denecektir.

UNDO ve REDO

Peki, UNDO ve REDO nedir?

Öncelikle PostgreSQL 8.0'dan önceki durumu inceleyelim ve WAL kavramını açalım:

Üstte de belirttiğimiz gibi, 7.1 sürümü ile birlikte PostgreSQL'e WAL (Write Ahead Logging, xlog olarak da bilinir) eklenmiştir. PostgreSQL'de transactionlar, o transaction commit edilmeden hemen önce diskte transaction loguna yazılırlar (xlog, WAL). Bu dosyaların boyutu ön tanımlı olarak 16 MB'tır (bu değer derleme aşamasında değiştirilebilir). WAL sayesinde disk işlemlerinin sayısı azalmaktadır; çünkü diskteki veri her transactionda değil; belirli aralıklarla değişir. WAL yapılandırması ile ilgili ayrıntıları kitabımızın “Başarım” ile ilgili bölümünde bulabilirsiniz.

xloglar, bilinen bir başlama noktasından veri kurtarmak için gerekli olacak tüm bilgileri sağlar. Bu işleme REDO işlemi denir. Bu bilinen başlama noktaları, biten transactionların kaydedildiği ve clog adı verilen yerde tutulur. Cloglar transactionlar commit edildikten hemen sonra diske yazılırlar. Xloglar pg_xlog, cloglar da pg_clog dizini altında bulunur.

PostgreSQL'de değişen veri sayfaları (data pages) anında diske yazılmazlar; çünkü xlog ve clog içindeki bilgiler verinin kurtarılması zaman gereken her şeye sahiptir. Belirli zamanlarda, değişmiş (dirty-kirli olarak da adlandırılır) veri sayfalarının diske yazılması için checkpoint işlemi gerçekleştirilir. Checkpoint işlemi tamamlandığında, xlog içine son transaction numarasını bir

işaretçi olarak yazar ve clog dosyalarını son transaction idsine kadar ilerletir. Değiştirilmiş veri sayfalarının diske yazılması, bgwriter (lazy writer olarak da bilinir) aracılığı ile gerçekleştirilir. bgwriter süreci arka planda gerçekleşir; böylece yoğun veritabanı sunucularında checkpoint işlemlerinin etkisi azaltılmış olur.
Veri kurtarmada, veritabanı dosyalarının eksiksiz olduğu kabul edilir; ancak bu dosyaların güncel olmasına gerek yoktur. postmaster süreci tekrar çalışmaya başladığında, clog içinden arama yapar ve checkpoint yapılan son transaction numarasını bulur. Bu veriyi kullanarak, mevcut xlog dosyaları arasında bir arama yapar. Eğer clog içindeki id, xlog içindeki id'den küçükse, postmaster süreci kalan transactionları da REDO işleminden geçirir ve böylece sistem olası olan en uygun hale gelmiş olur.

Eğer uygun xlog dosyaları yoksa, veri kurtarmak mümkün olamayacaktır.

Initdb sürecinden hemen sonra en az 1 tane xlog yaratılır. xlog içine yeni veriler yazıldıkça, gerekli oldukça yeni xlog dosyaları yaratılacaktır. Checkpoint işleminin sonucu olarak, checkpoint işleminden sonra xlogların gerek duyulmadığı zamanlar olacaktır. Her bir checkpointte, eğer artık gereksinim duyulmayan bir xlog bulunursa, bu xlog ya kaldırılacak ya da bu xlogun üzerine yeni veriler yazılacaktır. xloglar “sıranın” önüne getirileceklerdir; böylece veritabanı yöneticileri bu dosyaları silmek ya da kaldırmak zorunda kalmayacaktır. Belirli miktarda dosya önceden ayrılmış loglar olarak saklanacaklardır; bu miktar önceden kontrol edilebilmektedir. Bu sınıra erişildiğinde, xloglar yeniden başa döndürülmek yerine silineceklerdir. Bunun sonucu olarak, xlogların sayısı zaman içinde değişebilir.

Eğer bir xlog, ilgili disk bölümünde boş yer kalmaması nedeniyle yazılamazsa, xloga bağlı olan transaction (ya da ona bağlı olan diğer tranactionlar), disk sorunu çözülene kadar commit edilemeyecektir. Mevcut durumda bu durum transactionları, pg_xlog dizinindeki yer kadar sınırlamaktadır.

Xloglar göreceli olarak cloglara göre çok daha fazla yer kaplarlar. Clog dizini içinde boş yer kalmaması durumu beklenen birşey değildir.

İşte bu üstte anlatılanlar REDO işlemidir. PostgreSQL 7.1'den 8.0'a kadar geçen sürede üstteki bilgiler geçerli idi. 8.0'dan sonra REDO işleminin yanı sıra UNDO işlemi de PostgreSQL'e eklenmiştir.

Şimdi başarısız olabilecek durumların bir analizini yapalım:

- Bir transaction başarısız olursa, xlog'a birşey yazılmaz ve clog girdisi başarısız olan son transaction id' sini gösterir.
- Eğer bir transaction başarılı olursa, değişiklikler xlog'a kaydedilir ve clog girdisi başarılı olan son transaction id'sini gösterir.
- Eğer xlog dizini dolarsa ya da başka bir nedenden dolayı yazılamıyorsa (dizin izinlerinin değiştirilmesi gibi...) PANIC hatası verilir.
- Eğer clog dizini dolarsa ya da başka bir nedenden dolayı yazılamıyorsa (dizin izinlerinin değiştirilmesi gibi...) PANIC hatası verilir.

Point-In-Time-Recovery (PITR)

Point In Time Recovery, PostgreSQL'de önceki sürümlerde olan veri kurtarma tekniklerini geliştirmek amacıyla getirilmiş bir özelliktir. Bir çok kurumsal veritabanında da PITR bulunmaktadır. Bu özellik sayesinde çökme anında veri kurtarmanın olası olamayabileceği durumlar en aza indirilir. Bu durumları şu şekilde özetleyebiliriz:

- Veritabanı nesneleri silinmiş olabilir (Örnek: DROP DATABASE)
- xlog'lar, REDO işlemi için yeteri kadar geriye gitmiyor olabilir.
- Veritabanı dosyaları eksiksiz değildir ve REDO işleminden önce tamamen değiştirilmelidir.

Bu durumlarda, veriyi kurtarabilmek için sisteminizin tam fiziksel yedeğinin olması gerekir.

Tablespaceler sayesinde, tablespaceleri teker teker yedekleyip onları kurtarmak mümkündür. Ayrıca xlogların normal xlog dosya sisteminden çıkartılıp başka bir yere yedeklenmesinin sağlanması gerekir.

PITR ayrıntıları

PITR' nin gerçekleşebilmesi için, belirli bir zaman diliminde PostgreSQL'in çalışma anında alınmış eksiksiz bir fiziksel yedeği bulunmalıdır. Bu yedek tüm veri ve clogları, ayrıca (varsa) kısa yollarla belirtilmiş tüm tablespaceleri içermelidir. Ayrıca, tüm xloglar da **sıralı** olarak elinizde bulunmalıdır.

PITR çözümü iki temel işlemden oluşur:

1. xlog arşivlenmesi
2. Belirli bir zamana kadar kurtarma işlemi (Recovery-to-point-in-time – RPIT)

1. xlog arşivlenmesi

Verinin kurtarılabilmesi için, üstte de yazıldığı gibi, xlogların elimizde olması gerekmektedir. PostgreSQL PITR API' si, tüm logların arşivlenmesi gerektiğini kabul eder. Böyle koşulsuz olarak tüm xloglar, postgresql.conf dosyasında belirtilecek şekilde yedeklenir.

2. Belirli bir zamana kadar kurtarma işlemi (Recovery-to-point-in-time – RPIT)

Verinin belirli bir zamana kadar kurtarılması, aşağıdaki seçenekleri bize sunmaktadır:

- xlogların sonuna kadar veri kurtarma
- Mevcut tüm xlogların sonuna kadar veri kurtarma
- Belirtilen ana kadar olan verilerin kurtarılması
- Belirtilen transaction id'sine (xid) kadar kurtarma

Son iki işlem UNDO işlemidir.

Bunlardan birincisi zaten PostgreSQL içinde olan bir özellikti (yukarıda açıklanmıştı).

Bunların gerçekleşebilmesi için en temel kural şudur: “PostgreSQL'i archive modda başlatın ve temel yedeğinizi (base backup) alın.”

Veritabanı sunucusu archive modunda iken çökerse, daha önceden olduğu şekilde (üstteki ilk madde) veriyi kurtaracaktır. Eğer ana sistem çökerse, ya da başka bir nedenle veri kurtarmaya gereksinim duyulursa, archive recovery modu aşağıdaki süreçlerle tetiklenebilir:

1. Daha önceden alınmış olan eksiksiz veritabanı yedeğinin geri yüklenmesi
2. $PGDATA dizininin içinde recovery.conf dosyasının uygun içerikle oluşturulması
3. PostgreSQL' in yeniden başlatılması

Kurtarma sürecinin başarılı olması için aşağıdakilerin mutlaka sağlanması gerekir:

1. PostgreSQL içindeki her bir dosyanın tam yedeğinin alınması (eksiksiz olması sürecin gerçekleşebilmesi için çok önemlidir).
2. Temel yedeğin alınmasının hemen ardından başlayan ve kurtarma işleminin sonlanacağı ana kadar olan verileri içeren, arşivlenmiş WAL logları.

Eğer bu iki maddeden birisinde küçük de olsa bir eksiklik olursa, sadece bu zincirin kırıldığı yere kadar olan kısmın kurtarılabileceğini lütfen unutmayın.

PostgreSQL size xlog yedeklemesinde hiç bir kısıtlama getirmez. Bu parametre içerisine, o işletim sisteminde kullanılabilecek herhangi bir komut verilebilir.

PITR Kullanımı

PITR kullanımı ve hazırlanması genel olarak çok kolaydır. Süreçlerin sırayla takip edilmesi yeterlidir.

WAL Arşivleme

PITR için 1. aşama WAL arşivlenmesidir. PostgreSQL size bu konuda herhangi bir kısıtlama getirmez. İşletim sistemi tarafından desteklenen herhangi bir komutla yine işletim sistemi tarafından erişilebilen herhangi bir kayıt ortamına xloglar arşivlenebilir – hatta aynı anda birden fazla yere arşivlemek de mümkündür. WAL arşivlenmesi etkin olmadığı zamanlarda PostgreSQL, belirtilen sayıda WAL segmentine ulaşınca en başa döner ve ilk dosyaya tekrar yazmaya başlar. Tabii ki bu işlemden önce, PostgreSQL tarafından `CHECKPOINT` yapılır ve diskteki veriler güncellenir.

WAL arşivlemesini etkin kılmak için `postgresql.conf` dosyasındaki `archive_command` parametresinin değiştirilmesi yeterlidir:

```
archive_command = 'cp -i %p /home/gdg/pgarchivedir/%f </dev/null'
```

%p arşivlenecek dosyayı gösteren makrodur. Örneğin:

```
       /var/lib/pgsql/data/pg_xlog/000000010000000000000090
```

%f ise dosyanın adıdır: `000000010000000000000090`

`archive_command` ile belirtilen komut, `postgres` kullanıcısının hakları ile çalışır. Bu nedenle, belirtilen dizine `postgres` kullanıcısı yazabilmelidir -- ve bu dizini sadece `postgres` kullanıcısı okuyabilmelidir. Burada belirtilen dizinin dolması durumunda, sorun çözülene kadar xloglar `pg_xlog` dizini altında tutulur. Daha önce de belirttiğimiz gibi `pg_xlog` dizini de dolarsa PostgreSQL kendisini durduracaktır. Bu özellik veri bütünlüğü için gereklidir.

WAL arşivlemesi, ilgili xlog dolduktan sonra gerçekleşecektir. Bu nedenle, çok az yoğun olan sistemlerde xlog arşivlemesi gecikebilir. Bu sistemlerde, `archive_timeout` parametresi ile sunucunun xlog dolmasa bile xlogu arşivlemesi sağlanabilir. Bu değer ön tanımlı olarak 0'dır ve saniye cinsindendir – yani arşivleme için xlogun dolması beklenir. Bu değerin çok kısa tutulması disk ve ağda yoğunluk yaratacaktır – bu nedenle genel olarak bu sürenin 60 saniyeden az olmaması önerilir.

`archive_command` parametresindeki değişikliklerin etkin olması için PostgreSQL'in yeniden yüklenmesi (reload) yeterlidir.

Temel yedek alma (base backup)

PITR için 2. aşama temel yedeğin alınmasıdır. Bu işlemin yapılmadan önce üstteki bölümde anlatılan WAL arşivlemesinin etkin olduğundan emin olunuz.

Ardından veritabanı sunucusundaki herhangi bir veritabanına postgres kullanıcısı ile bağlanın (ya da superuser yetkili başka bir kullanıcı) ve `SELECT pg_start_backup('ET▪KET');` ile süreci başlatın. Bu komut, `$PGDATA` altında `backup_label` adında bir dosya oluşturur. Verdiğiniz etiket bu yedek için özgür bir ad olmalıdır Örnek bir komut ve çıktısı şu şekildedir:

```
-bash-3.2$ psql gdg -q
gdg=# SELECT pg_start_backup('gdg');
 pg_start_backup
-----------------
 0/93000020
(1 row)
```

Şimdi de bu `backup_label` dosyasının içeriğine bakalım:

```
START WAL LOCATION: 0/93000020 (file 000000010000000000000093)
CHECKPOINT LOCATION: 0/93000020
START TIME: 2007-12-27 02:01:16 PST
LABEL: gdg
```

İPUCU: Eğer bu komuttan sonra bir kez daha pg_start_backup() çalıştırılırsa, PostgreSQL hata mesajı verecektir:

```
pgkitap=# SELECT pg_start_backup('gdg');
ERROR:  a backup is already in progress
HINT:  Run pg_stop_backup() and try again.
```

Şimdi PostgreSQL veri dizininin ($PGDATA)yedeğini alabilirsiniz. Bunun için tar ya da benzeri sıkıştırma programlarını kullanabilirsiniz. Bu aşamada `pg_xlog` dizini dışındaki tüm dosyaların yedeklenmesi gerektiğini unutmayınız – yani pg_xlog dizini dışındaki herhangi bir dosya/dizini yedek dışında bırakmayınız. Tablespace kullanıyorsanız, onların da yedeğini almanız gerektiğini unutmayın. Örnek olarak, `tar --exclude $PGDATA/pg_xlog ..... $PGDATA` komutunu verebiliriz.

Yedekleme işlemi bittikten sonra, süreci bitirelim:

```
gdg=# SELECT pg_stop_backup();
 pg_stop_backup
----------------
 0/94000000
(1 row)
```

Bu komut, ayrıca, bir xlog geçiş işlemini de yapar (`pg_switch_xlog()`) Bu komuttan sonra WAL arşivlemesinin yapıldığı dizinde sonu .backup ile biten bir dosya oluşturulur ve bir önceki komutta yaratılan dosya silinir:

```
START WAL LOCATION: 0/93000020 (file 000000010000000000000093)
STOP WAL LOCATION: 0/94000000 (file 000000010000000000000094)
CHECKPOINT LOCATION: 0/93000020
START TIME: 2007-12-27 02:01:16 PST
LABEL: gdg
STOP TIME: 2007-12-27 02:16:57 PST
```

pg_start_backup ve pg_stop_backup arasında ne kadar sürenin geçtiği ya da ne kadar transaction'ın olduğu önemli değildir.

Artık temel yedeğiniz hazır. Ancak, WAL arşivlenmesinin düzgün yapılabildiğini düzenli olarak kontrol etmeniz önemlidir.

WAL arşivinden geriye dönme (PITR)

İdeal dünyada herşeyin yolunda gitmesini beklerken, veritabanı yöneticisinin yanlışlıkla DROP DATABASE/TABLE/vs komutunu çalıştırması, ya da binlerce kaydı yanlışlıkla silmesi, ya da benzeri işlemleri yapması mutlaka başa gelebilecek bir olaydır. Benzer şekilde diskini artık çalışmayı durdurmuş olabilir. Benzeri başka şeyler de olabilir. PITR sayesinde bu durumdan kurtulabiliriz:

- Öncelikle (eğer hala çalışıyorsa) PostgreSQL süreçlerini durdurun. Ardından, daha önceden aldığınız $PGDATA yedeğini (temel yedek) açınız ve $PGDATA altına taşıyınız. Varsa tablespaceleri de yerlerine açınız.

- pg_xlog dizinini temizleyiniz (eğer bu dizini de yedeklediyseniz). Bu dizini yedeklemediyseniz pg_xlog dizinini yaratınız. Eğer bir şekilde mevcut ise, arşivlenmemiş xlog dosyalarını da pg_xlog altına kopyalayınız.

- Şimdi recovery.conf dosyasını yaratabilisiniz. Bu dosyanın bir örneği PostgreSQL kodu ya da paketleri ile birlikte gelmektedir. Bu dosya $PGDATA altında olmalıdır. Örnek bir recovery.conf dosyası aşağıdaki gibidir:

```
restore_command = 'cp /homeUgdg/pgarchivedir/%f %p'

Bu parametre mevcut tüm xlogların okunması için yeterlidir. Ancak başka parametrele
de mevcuttur:

#recovery_target_time = '2007-12-26 22:39:00 EST'
Belirli bir tarihe kadar olan transactionlar i▪lenir.

#recovery_target_xid = '1100842'
Belirli bir transaction idsine kadar olan transactionlar i▪lenir.

#recovery_target_inclusive = 'true'
Üstte belirtilen de▪erin kurtarma i▪lemine dahil olup olmayaca▪ını belirtir.

#recovery_target_timeline = '33'
pg_control'da gösterilen ana timeline dı▪ındaki ba▪ka bir timeline'a geri dönmek
isterseniz bu parametreyi kullanabilirsiniz. Buraya bir sayı ya da bir 'latest'
yazmak gerekir. Öntanımlı de▪eri latest olarak belirlenmi▪tir.
```

- PostgreSQL'i başlatın. postmaster, `recovery.conf` dosyasını okuyup gerekli işlemleri bitirdikten sonra bu dosyanın adını `recovery.done` olarak değiştirir. Böylece sunucunun beklen(me)yen bir yeniden başlaması durumunda kurtarma süreci de yeniden başlamaz.

- Bazı durumlarda kurtarma işlemine başlamadan önce pg_hba.conf'tan erişimlerin kapatılması gerekebilir. Böyle bir durumda kurtarma işlemi bittikten sonra erişim dosyasını eski haline getirmeyi unutmayınız.

İPUCU: Eğer belirtilen sınıra kadar bir xlog bulunamazsa, sıralı olan xlogların sonuna kadar kurtarma işlemi yapılır.

İPUCU: Eğer xloglar sıralı değilse, kurtarma işlemisıranın bozulmasından önceki son xloga kadar yapılır.

İPUCU: WAL logları eğer oluşturulma hızından yavaş arşivlenebiliyorsa, pg_xlog dizini dolduğunda PostgreSQL çalışmayı durduracaktır. Arşiv hızının yeterli olduğuna emin olunur.