『SQLアンチパターン 第2版』日本語版付録

砂の城

奥野 幹也

> 地震・津波による被災の可能性、自然現象を起因とするシビアアクシデント（過酷事故）への対策、大量の放射能の放出が考えられる場合の住民の安全保護など、事業者である東京電力（以下「東電」という）及び規制当局である内閣府原子力安全委員会（以下「安全委員会」という）、経済産業省原子力安全・保安院（以下「保安院」という）また原子力推進行政当局である経済産業省、（以下「経産省」という）が、それまでに当然備えておくべきこと、実施すべきことをしていなかった。
>
> ——国会事故調調査報告書（http://naiic.go.jp/report/）より

2011年3月11日、東日本を大地震が襲いました。震災による被害は甚大なものでしたが、それに拍車をかけて事態を悪化させたのが過去最悪の原発事故です。この事故により、福島には広大な立ち入り禁止区域ができてしまいましたし、破損した原発の撤去（修復？）は10年以上経過した今も終わる気配がありません。原発周辺での地震の規模も確かに大きかったのですが、本来はこの程度の地震で倒壊すべきではありませんでした。国会事故調調査報告書によれば、原発事故は「人災」、すなわちヒューマンエラーです。ヒューマンエラーが、ここまでも大きな被害をもたらす結果になってしまいました。

ヒューマンエラーによって大きな被害がもたらされるといえば、ITの分野も負けてはいません。データ消失やデータ漏洩によって甚大な被害がもたらされるインシデントがたびたび発生しています。皆さんの中にも痛い経験をされたことのある方がいらっしゃるのではないでしょうか。周りを見渡しても、たびたびサービス停止やデータ消失、あるいは漏洩などのアクシデントが報じられています。明日は我が身です。油断しているといつなんどきそのような被害の渦中に巻き込まれることになるかも知れません。

幸いにして、ITにはそのようなインシデントからシステムを守るためのテクニックが多数存在します。ただしそれらのテクニックが活かしきれていないことも見受けられます。

目的：サービスの安定稼働

今日では、様々なサービスが24時間365日の連続稼働を前提で提供されています。基幹系業務は無

停止が基本ですが、ユーザーが無料で利用できるウェブサービスやスマホアプリですら無停止が当たり前のようになっています。サービスを提供する側にとって、サービスの停止は収益の減少に直結するからです。たとえブログやゲームなどの無料サービスであっても、データ消失などが起きれば信頼が大きく失墜することでしょう。また、サービス停止中にユーザーが離れてしまうといった懸念もあります。

収益の減少を極力減らすには、様々なテクニックを駆使して停止時間を極限まで抑えることです。手法によってコストは様々ですので、停止によって失われると予測される収益の規模によって手法を使い分けます。

アンチパターン：想定不足

問題は、どのようなことが起きるかという想定と、それぞれの事象への対策が十分に検討されていないことです。コンピュータシステムにおいて安全神話はありません。サービスを安定稼働させるには、トラブルは当然起きるものとして想定しておく必要があります。想定しただけでは不十分で、実際に運用時に事象が起きたときどのような対応をするべきかといった検討も必要です。どのような種類の問題が起きるかという点については、マシンの停止やトランザクションの失敗といった定石を抑えるのはもとより、事象を網羅的に想定するには過去に起きた事例が大いに参考になるでしょう。事象に応じて自動的にサービスを継続させる仕組みや対応手順を決めておくことで、被害が最小限に抑えられます。

意外となされていないのが、性能問題や障害が起きた時の対処をどうするかというポリシーの策定です。そういった問題に対処するには種々のデータが必要となりますが、データの採取を行なうと、運用中のシステムに対してディスクスペースやオーバーヘッドといった負担がかかることがしばしばです。あらかじめ、どこまでのデータ採取なら許容できるかといったことを決めておけば、問題の対処もスムーズに行えます。反対に、採取することになっていないデータが必要な場合には、原因の究明を諦めて回避策に注力すると言った対応が求められます。

データ採取やリカバリ処理などは、日頃から訓練しておくと良いでしょう。実際にやってみると手順を間違ったり想定以上の時間がかかったりするものですが、実際にやってみればそのような問題も洗い出すことができます。自動で回復するような仕組みを実装している場合には、それを試験しておくようにしましょう。被害の拡大を防ぐためにもそういった訓練や試験は無駄にはなりません。備えあれば憂いなしです。

アンチパターンの見つけ方

次のようなセリフを耳にしたら、想定すべき状況を見落としているかも知れません。

「データサイズが一ヶ月で3倍になった」

想定以上の早さでデータが増加するのはよくあることですが、それは想定外の事象が起きやすい状況です。想定を超えたデータサイズでの運用はすでに未知の世界です。いつなんどきサービスへ深刻な影響を及ぼす事象が起きても不思議ではありません。想定以上のアクセスやユーザー数でも状況は同じです。

「データベースへの更新でデッドロックが起きる。製品のバグじゃないか？」

知識不足は大敵です。製品や技術への理解なくして適切な想定はできません。ちなみに、デッドロックは行単位で排他制御を行うRDBMSなら必ず発生し得る状況であり、適切な例外処理（主にトランザクションのリトライ）を実装しておく必要があります。場合によってはユーザーに処理のやり直しを求める必要があるでしょう。

「問題を解決するためにベンダーが要求しているデータは本番環境の負荷が高すぎるので採取できない」

サービスがカットオーバーしてから常に順風満帆で何事もなく運用できるなどと考えてはいけません。問題は当然起きるものであり、問題が起きたらデータの採取が必要となります。そして、データ採取にはオーバーヘッドが付きものです。欲しいデータが詳細であればあるほど、オーバーヘッドは大きくなります。使用しているRDBMS製品ではどのようなデータを採取することができ、それぞれどの程度のオーバーヘッドがあるかを事前に調べておくべきです。採取時にサービスへの影響が出るようなデータが必要になったときにどうするべきか。いざ必要となったときに慌てないよう、事前に想定しておくことが必要です。

「最近マシンをアップグレードした。だから性能の問題とは無縁だろう」

マシンを新しくしたからと言って、性能の問題から永遠に逃れられるわけではありません。確かにスペックの高いマシンを使用すると一時的に性能は良くなりますし、その効果は非常に大きなものです。ただし、CPUやメモリを増強しても、ボトルネックがディスクI/Oやネットワークでは意味がありませんので、どのリソースが不足しがちかということをよく監視しておくことが重要です。システム設計時には想定していなかったボトルネックが後から顕在化することもしばしばですから、状況に合わせてマシンを強化することも運用には必須です。「いったん構成を決めたら終わり」ではなく、「将来的にマシンの増強が必要になるかも知れない」と想定しておくのが現実的でしょう。また一方で、マシンを強化したところで、その効果はたかが知れているとも言えます。CPU速度やメモリの容量が増えたとしても、多くて数倍程度でしょう。非効率なクエリや突発的なアクセスといった事象による負の効果は、数倍などというオーダーの性能アップでは到底安心できるものではありません。マシンの増強は万能ではないということも、常に念頭においておきましょう。

アンチパターンを用いてもよい場合

対策を講じれば講じるほどコストはかさみますので、無計画に対策を講じれば良いということではありません。仕組みや体制を考えるのは時間がかかる作業です。どこまでお金や手間をかけられるかは、そのシステムが停止することでどれだけの損失があるか、もしくは事業計画に影響があるか、どの程度の停止時間なら許容できるかといったことを考慮して検討する必要があります。オーバースペックにならないよう気をつけましょう。

スモールスタートで始めたウェブサービスなどではほとんど対策がなされていない場合がありますが、後から抜本的なアーキテクチャの見直しが必要になるかも知れません。X（旧Twitter）やFacebookのような大規模サイトが何度もアーキテクチャの見直しを迫られているのは有名な話です。規模に応じて最適なシステムアーキテクチャは異なりますので、規模が大きくなるにつれて形を変えていくのは当然の話です。規模が大きくない時点で大規模サイトのまねをしても良いことは何もありません。

解決策

重大なインシデントを回避し、サービスの安定稼働を目指すのに必要なのは、どのようなトラブルが起こりうるかということを可能な限り想定しておくことです。トラブルは必ず起こります。壊れないハードウェア、バグのないソフトウェア、脆弱性のないアプリケーションという都合の良いものはありません。トラブルを「当然起きる日常的なもの」として捉えることが必要です。

ここでは、サービスの運用を始めるにあたって実施・想定しておくべき代表的な対策について紹介します。

ベンチマーク

大きなトラフィックが予想されるシステムでは、事前にどの程度まで処理が可能なのかということをベンチマークしておきましょう。運用を続けていくうちにトラフィックやデータ量の増大によって性能の問題に突き当たるシステムは多く、まさに「石を投げれば当たる」状態です。トラフィックやデータ量の増大によって性能の限界に突き当たるのは「避けられない」現象です。水を入れ続ければコップがいっぱいになってあふれてしまうのと同じです。大切なのは、限界がいつやってくるのかということを予測し、限界が来る前に対策をとることです。ボトルネックになりやすいのはデータベースだと言われて久しいですが、限界がコントロールできていれば怖くはありません。クラウド上でデータ量や処理量に応じて自動的に拡張できるようなサービスもありますが、性能の限界を突破できても、料金も予算の天井を突破してしまっては困りますよね。

性能の問題を回避するための第一歩がベンチマークを取ることです。どの程度のトラフィックやデータ量まで耐えることができるかということを知っていれば、それらが増加するペースと照らし合わせる

ことで、あとどのぐらいで限界を迎えるかという予測がある程度できます。限界が来てから慌てて対策をするのではなく、対策は計画的に行うべきです。ときにはアーキテクチャの変更が必要になる場合があるからです。（例えばシングルサーバーからレプリケーションを用いたスケールアウトやシャーディングへの転換など。）

ベンチマークをするにあたって重要な点は次の3つです。

1つめは、現実に即したシナリオを用いることです。ベンチマークはアプリケーションがどの程度の負荷まで耐えることができるかということを調べるためのものですから、アプリケーションと同じ負荷をかけなければ意味のある数字をはじき出すことはできません。ウェブアプリケーションであれば、ウェブアプリケーション用のロードジェネレーターを使い、現実に即したシナリオで負荷テストをするべきです。

2つめは、実際のアプリケーションで用いられるのと同程度のデータサイズで実施することです。データサイズが小さいうちは、データがすべてキャッシュ（InnoDBバッファプール）に収まるために高速で、データサイズが大きくなってから急激に性能が低下してしまうというのはよくあることです。十分に大きなデータを使ってベンチマークを行いましょう。

最後に、当然のことですが実際のシステムと同じハードウェアやOSを使いましょう。ベンチマーク用にテスト環境の型落ちのマシンを使うのはよくありません。ベンチマーク結果は使用するシステムによって大きく異なります。実際に使用するのと同じシステムでなければ、ベンチマークによって得られた数値には意味がありません。CPUの動作周波数が1.5倍違うからベンチマークの結果も1.5倍の差になるというふうにはならないのです。

テスト環境の構築

サービスをリリースしてからが本当のデバッグの始まりです。アプリケーションのデバッグはもちろんのこと、利用しているミドルウェアやデータベース管理システムのトラブルシューティングでもテスト環境は大活躍します。特に問題の切り分け（アプリケーションの問題なのか、それともミドルウェアやデータベースの問題なのか）や、本番環境では採取できない詳細なデータを採取する、本番環境に対する変更作業をリハーサルするといった用途には必須です。ベンチマークシナリオがよくできていれば、性能の問題でも活躍します。

テスト環境に用いるシステムは、本番環境で利用しているものと同じものを1セット用意するのが理想です。アプリケーションの修正を行った場合に性能が劣化してしまわないかを調べるため事前にベンチマークをするといった用途には、そっくり同じシステムでなければ意味がありません。有事に備え、テスト環境を事前に準備しておく、またはすぐに利用できる体制をとっておくようにしましょう。そういう意味ではパブリックかプライベートかは問わず、クラウド環境はコスト面で有利かも知れません。

例外処理

データベース管理システムを用いたアプリケーションでは適切な例外処理を実装することが必須です。適切な例外処理は、「どのような例外が発生し得るのか？」という想定ができていることの裏返しです。そのためには、トランザクション実行中にどのようなエラーが発生するかを知っておく必要があります。

例えばトランザクションのデッドロックや（ロック待ちの）タイムアウト、重複キーエラー、外部キーエラーなどはどの製品でも起きるエラーであり、トランザクションをリトライする例外処理を仕込んでおくのが一般的です。その他、データベースサーバーへの接続が切れた場合の対処や、データベース製品固有の一時的な（致命的でない）エラーへの対処が必要となります。製品固有のエラーにもれなく対処するには、その製品に精通している必要があります。どのようなエラーが起きるかを想定するには製品知識は重要です。

構文エラーのように、明らかにバグまたはSQLインジェクションに対する脆弱性の可能性を示すような致命的なエラーが発生した場合には、即座に関係者へ連絡が行くような仕組みを作っておくと良いでしょう。

バックアップ

サービスの最後の生命線がバックアップです。ディスク装置の故障やオペミス、クラッカーによる攻撃などによって本番環境のデータが破壊されるというインシデントは起こりえます。データが破壊されればもちろんサービスは止まってしまいますし、肝心のデータが破壊されてしまっていては復旧すらもできませんので致命的です。そんなとき、最後の防衛線となるのがバックアップです。この防衛ラインを越えたら後はありません。バックアップは万全にしておきましょう。

バックアップにまつわる誤解の代表格が、「RAIDで冗長化しているからバックアップはとらない」というものです。ディスクの冗長化はバックアップにはなりません。データが論理的に破壊されてしまうようなケースに対応できないからです。例えば誤って必要な行を`DELETE`した場合、ひとたびそれを`COMMIT`してしまったならば、いかにディスク装置を冗長化していようともデータは元に戻りません。`COMMIT`してしまったら元に戻せないのがRDBMSの特性です。（取り消せるならそれはトランザクションではありません！）RAIDだけでなく、レプリケーションもバックアップの代わりに使用することはできません。オペミスによる変更などが即座に伝播してしまうからです。即座に変更を伝搬しないようにする遅延レプリケーションといった技術もありますが、復旧の手順をきっちり整えておかないと、気づいたときにはすでに変更が伝搬されてしまっていて後の祭りになってしまいます。

バックアップ運用は万全に行いましょう。

高可用性

どれだけ高価なマシンを使おうと、マシンそのものの故障から完全に逃れることはできません。停止時間をできるかぎり短くするには、マシンを冗長化する仕組みを考えておく必要があります。マシンを冗長化しない場合、マシンが故障から復帰するまでサービスを再開することができないかも知れません。ハードウェアをまるごと交換するのでもない限り、切り分けには時間がかかります。運が良ければOSのsyslogにエラーの部位を示す兆候が出ていたり、POST等で被疑箇所を特定できることがありますが、延々と切り分けができないケースもあります。また、新しく交換したパーツが故障していることもあるでしょう。マシンを冗長化しない運用は、かなり停止時間が長くなることを覚悟しなければなりません。(最悪数日という場合もありえます。)

一般的に高可用性構成と言えば、クラスタリングソフトウェアを用い、マシンがクラッシュした際にフェイルオーバーさせるものを指します。この場合、フェイルオーバー時にサービスを引き継いだ方のマシンにおいてファイルシステムやデータベースのクラッシュリカバリが行われます。ログファイルのサイズが大きかったり更新が激しかったりすると、クラッシュリカバリには時間がかかるかも知れません。ディスクを冗長化しておくことも、ダウンタイムを短縮する上では有効です。

データベース製品によってはアクティブ/アクティブ構成やレプリケーションをサポートしているものがありますので、停止時間を短くするためにそういった機能を活用するのも重要です。コストと停止時間を天秤にかけて最適な構成を選びましょう。

ディザスタリカバリ

本当に重要なシステムでは、マシン単位の冗長化では十分でなく、データセンターないしはサイト全体の障害も考慮する必要があります。地震や津波などの自然災害、火災、電源の故障などにより、データセンターの一部または全体が使用不能になってしまった場合には、他のデータセンターで処理を引き継ぐというディザスタリカバリまで考慮しておくと良いでしょう。その場合には、データを継続的に複製し続けるレプリケーションなどの仕組みが必要になるでしょう。ただし遠隔地のデータベースとリアルタイムで同期を取ることは諦めたほうが良いでしょう。地理的に離れたマシンとの通信には、どうしても無視できないタイムラグが生じるからです。遠隔地には、非同期でデータを複製するような構成を検討しましょう。

運用ポリシーの策定

様々なテクニックを駆使して、問題が生じても自動でサービスを継続できる仕組みを構築することは素晴らしいですが、すべての事象をカバーできるわけではありません。システムの限界を超えた事象というのはどうしても起こりえます。

高可用性の限界を超えた障害

クラスターで二重化していたマシンが全滅したり、データセンターの特定のエリアの電源が全滅するというように、準備してある高可用性要件の限界を超えた場合にどうするべきかというのは1つの課題です。共有ディスクが故障してデータへアクセスできなくなる、ファイルシステムの損傷がひどくてfsckが終わらない、RDBMSのクラッシュリカバリが終わらないなどの要因で、フェイルオーバーが上手く行かない場合もあるでしょう。そのような場合、フェイルオーバーを繰り返すいわゆる「ピンポン」と呼ばれる現象が起きることになりますが、これはよく見かける風景です。そのような場合にどう対処をするかは現象次第ですが、ディスクを初期化してバックアップからリストアする、クラスターの使用をいったん取りやめて1つのマシンでサービスを起動するというような対処を念頭に置いておきましょう。

問題の調査

データベースと言えども、運用中に発生した問題の調査は厄介です。特に再現性がなく、トランザクションが失敗する程度の影響で、なおかつ稀にしか発生しないようなエラーは情報の採取も難しく、調査は難航することになります。本番環境から情報を採取するにしても、サービスへの影響を考えると詳細なデータが欲しくても手が出ないという状況にも陥りがちです。データベースソフトウェアには様々な情報採取の方法がありますが、オーバーヘッドが大きく、サービスへの影響があるものも少なくありません。影響があるからと言って一切手を出さないでいると、根本的な解決は永遠にできません。リソースに十分な余裕があるなら影響があっても許容できるはずですので、本番環境に対する影響がどの程度までなら許容できるのかを事前に検討しておきましょう。本番環境から情報を採取できないのであれば、テスト環境で再現を試みるしかありません。しかし再現性の低い問題を再現させるのは非常に手間がかかりますし、再現するかどうかははっきり言って賭けです。問題がサービスへ及ぼす影響が軽微である場合には、自動的にトランザクションをリトライして問題を無視するといった対処で留めておくことも視野に入れておくといいでしょう。

性能の劣化

処理に時間がかかるようになったり、マシンのリソースが限界まで消費されるような状況になった場合、最も効果的なのはクエリやスキーマをチューニングすることです。スケールアップやスケールアウトは、少しでもマシなクエリを書いてからにした方が良いでしょう。(NULLだらけのテーブルばかりといった、根本的に設計から見直す必要がある場合もありますが。)世の中には想像以上にひどい設計のスキーマや、ひどい効率のクエリがあふれています。きっとあなたのアプリケーションにもチューニングする余地があるはずです。クエリを適切にチューニングしても現在のアーキテクチャで性能の限界を迎えてしまった場合に、どのように性能を伸ばしていくかという戦略は重要です。現行のアーキテクチャでスケールアップやス

ケールアウトでどこまで性能が伸びるのか、抜本的にアプリケーションのアーキテクチャを見直す必要があるのか、その際RDBMS製品を入れ替えるのかと言った検討を事前にしておくようにしましょう。

どれだけ盤石な基盤を築けるかは、あなたがどれだけのインシデントを想定しているかが鍵なのです。

奥野 幹也（おくの みきや）
1975年生まれ。栃木県在住のギーク。フリー（自由な）ソフトウェアの普及をライフワークとしている。KDEを愛用。仕事ではMySQLのサポートに従事。著書には『エキスパートのためのMySQL[運用+管理]トラブルシューティングガイド』『MySQL Cluster構築・運用バイブル』（ともに技術評論社）、『詳解MySQL5.7：止まらぬ進化に乗り遅れないためのテクニカルガイド』（翔泳社）、『Pro MySQL NDB Cluster』（Apress）がある。

この日本語版付録は、初版和書『SQLアンチパターン』に収録した「25章 砂の城」を改訂したものです。