MySQL クラウド向け InnoDB チューニング

波多野 信広
株式会社インフィニットループ
2015/09/10

ver. 0.9.3

自己紹介

- 波多野　信広　(twitter @nobuHatano)
- 1969年札幌生まれ
- 東京で商用DBサーバーのサポートエンジニアを十数年
- 2012年にUターン
- （株）インフィニットループのインフラエンジニア
- ソーシャルゲームやWebサービスのMySQLと格闘して三年

MySQL クラウド向け InnoDB チューニング

アジェンダ

- オンプレミスな物理サーバーとクラウドの仮想サーバーとの違い
- クラウドでの MySQL サーバー構成例
- Linux や MySQL InnoDB の設定解説
- Cacti グラフを使っての実践的パフォーマンスチューニング
- まとめ

オンプレミスな物理サーバー

- スペックが高い部品を使って自由に構成出来る
- 電源、ディスク、等ハードウェアの障害確率高い
- ハードウェアの性能は一定している
- コンスタントな負荷も、バースト的な負荷も、両方可能
- 占有しているリソース（CPU,ストレージ）はコンスタントに負荷をかけるのが高スループットへ至る道
- 他者と共有しているリソース（ネットワーク）はバーストが得策

クラウドの仮想サーバー

- データセンター内に集積され耐障害性のあるPCサーバーがホストサーバー
- ゲストから見ると、電源停止など単純障害の確率は低い
- ハードウェアが低速になったりハングする等物理では見ない障害に遭遇
- 仮想は共有、つまりタイムシェアリングなので、I/O負荷はバーストが得意、コンスタントだと性能低下
- ディスクはエンタープライズな SAN (Storage Area Network) の RAID ストレージ

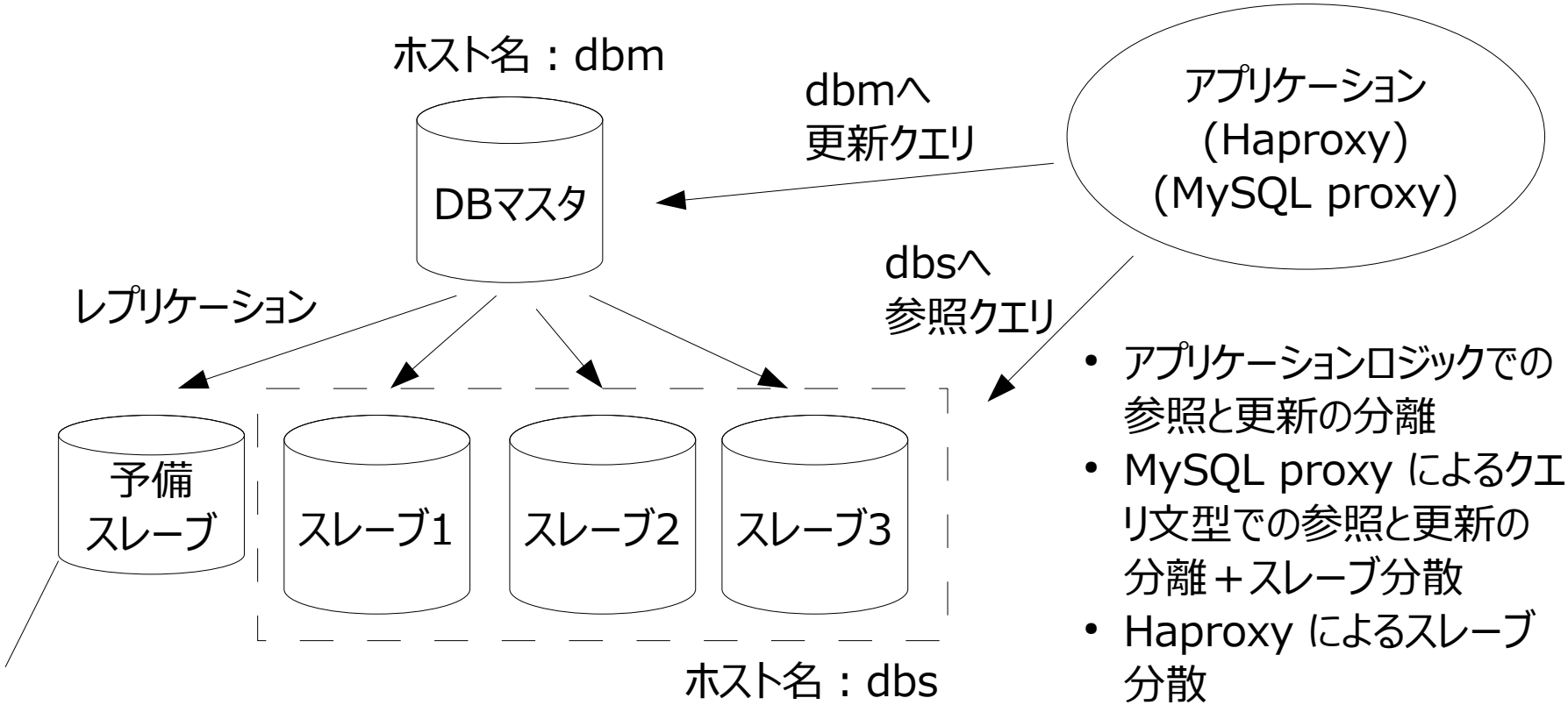
Linuxで SAN を使う課題：Write バリア

- Write I/O (fsync) はHDD内キャッシュへ書き込みで更新終了
  - DBには問題、Writeを毎回永続化するように指示する Write バリア
- Writeバリア有効だと低速
  - HDDはWriteスルーで毎回磁気メディアに書き込み
  - 素のランダムWriteになり大幅に低速
- SSDの寿命を伸ばす必要性 （＋高速化）に反する
  - RAIDコントローラーは冗長化しキャッシュはバッテリー保護
  - Writeを溜めることでSSD書き込み約半分にして長寿命化
  - 高速化も兼ねる
  - Writeバリア無効化が必要
- ではゲストの仮想サーバーなら Write バリアは？

物理と仮想の違い

- ストレージは物理と仮想で負荷のかけ方の戦略（バースト／コンスタント）から異なる
- 障害の発生頻度と種類が異なる
  - 物理：高頻度の単純障害　- ACID の (D) が試される
  - 仮想：低頻度だがハングアップなど特殊な障害 – 積極的なFailoverが必要
- ホスト側でのWriteバリア無効化 (データ保護を追及できない)
- 仮想サーバーを想定した割り切りが必要
- これらをふまえてシステムを構築

クラウドでの MySQL サーバー構成例



- 意図的に低性能のスレーブを予備に
- アプリからの参照に含めた場合は「坑道のカナリア」に
- スレーブを動的に追加する際の種に使える
- mysqldump取得に最適
- 別クラウド、別ゾーンを使うなど高信頼性追求
- KPI調査など高負荷OLAPなクエリも本番中実行可能

- アプリケーションロジックでの参照と更新の分離
- MySQL proxy によるクエリ文型での参照と更新の分離＋スレーブ分散
- Haproxy によるスレーブ分散
- 内部DNS (mydns, bind, etc..) によるスレーブ分散
- /etc/hosts によるスレーブ・ラウンドロビン

クラウドでの InnoDB チューニングとは

- 前述のような構成で MySQL のサーバー群を構成しチューニング
  - Read負荷はスレーブでスケールアウト可能
  - DBマスタの負荷、つまりWrite負荷はスケールアウト不可
- Writeの性能をいかに高めるか

Linux や MySQL InnoDB の設定解説

- Linuxの設定
  - vm.swappiness
  - /proc/[pid]/limits
- MySQLの設定
  - max_connections
  - innodb_flush_method, innodb_buffer_pool_size
  - innodb_log_file_size, innodb_adaptive_flushing
  - innodb_double_write
- 他の設定値は一般的なので割愛

vm.swappiness=1

- Kernel はあればあるだけメモリをバッファに使います （free での表示）

```
# free -m
             total       used       free
Mem:          7517       4451       3065
-/+ buffers/cache:        277       7239
```

- メモリが不足で返却されますが、バッファに更新データがあるとすぐ返せません

```
# vmstat -a
procs -----------memory---------- -
 r  b   swpd   free   inact active
 0  0      0 3125888 2614436 1684840
```

- （従来は）ユーザープロセスは Kernel とのメモリの取り合いに負けてスワップアウトされたくないので vm.swappiness=0
- Kernel 2.6.32-303 (CentOS 6.4+) から動作が変わり、0 にすると一切スワップされなくなりました。不足すると OOM Killer によってユーザープロセス、つまり MySQL が kill されます
- スワップは出来るだけ避けたいが、OOMは困るので、sysctl の設定では vm.swappiness=1 を使いましょう

オープン数 /proc/[pid]/limits

- ulimit や /etc/security/limits.conf は PAM経由でのログイン処理で適用
- デーモンプロセスは起動スクリプトで ulimit の設定が必要
- 稼働プロセスの実際の値は cat /proc/[pid]/limits で確認可能
- Max open files (ulimit -n 相当)
  - ファイル+ソケットの最大数。デフォルト 1024 は足りない。65535 など最大化
  - my.cnf の [mysqld_safe] open-files-limit で設定
- Max processes (ulmit -u 相当)
  - プロセス/スレッドの最大数。MySQL はコネクション毎に1スレッド生成するので max_connections の分が必要
  - デフォルトの 1024 で足りている（理由は後述）

心を強く保って max_connections<1000

- InnoDB のロールバックセグメント上のUndoスロットの上限1023 (MySQL5.1)、拡張され130944 (MySQL5.5+)
- クラウドの普通のサーバーでは InnoDBの同時実行トランザクションは1000が実用的な上限
- 厳密にはコネクション ≠ トランザクション、DBマスタへのアクセスなので コネクション数 ≒ トランザクション数
- デフォルトの max_connections=151 で十分実用的
- では、何故、そしていつ増やしてしまうのか？

Connection 不足の原因

- “Too Many Connections” 接続エラーが出ると、max_connections が少な過ぎるので引き上げるべき、とインフラエンジニアは集中砲火を浴びますが
- Connectionが不足する原因
  - 問題クエリの実行やプラットフォーム（ホスト）障害で MySQL が遅延またはハング状態になったとき
  - スクリプト言語で持続的接続を使う場合、明示的にはcloseさせないので、プロセスの寿命とのバランス次第で Idle なコネクションが滞留する
    - （PHPの場合）LBの特性、Apache の KeepAlive, MaxClients, RequestsPerChild, PDO の Timeout, MySQL の wait_timeout でバランス調整
  - 同時に長時間トランザクションを多数実行で >1000
- いずれも max_connections だけ >1000 にしても解決にならない
- max_connections を決めたのでメモリの見積りに入ります

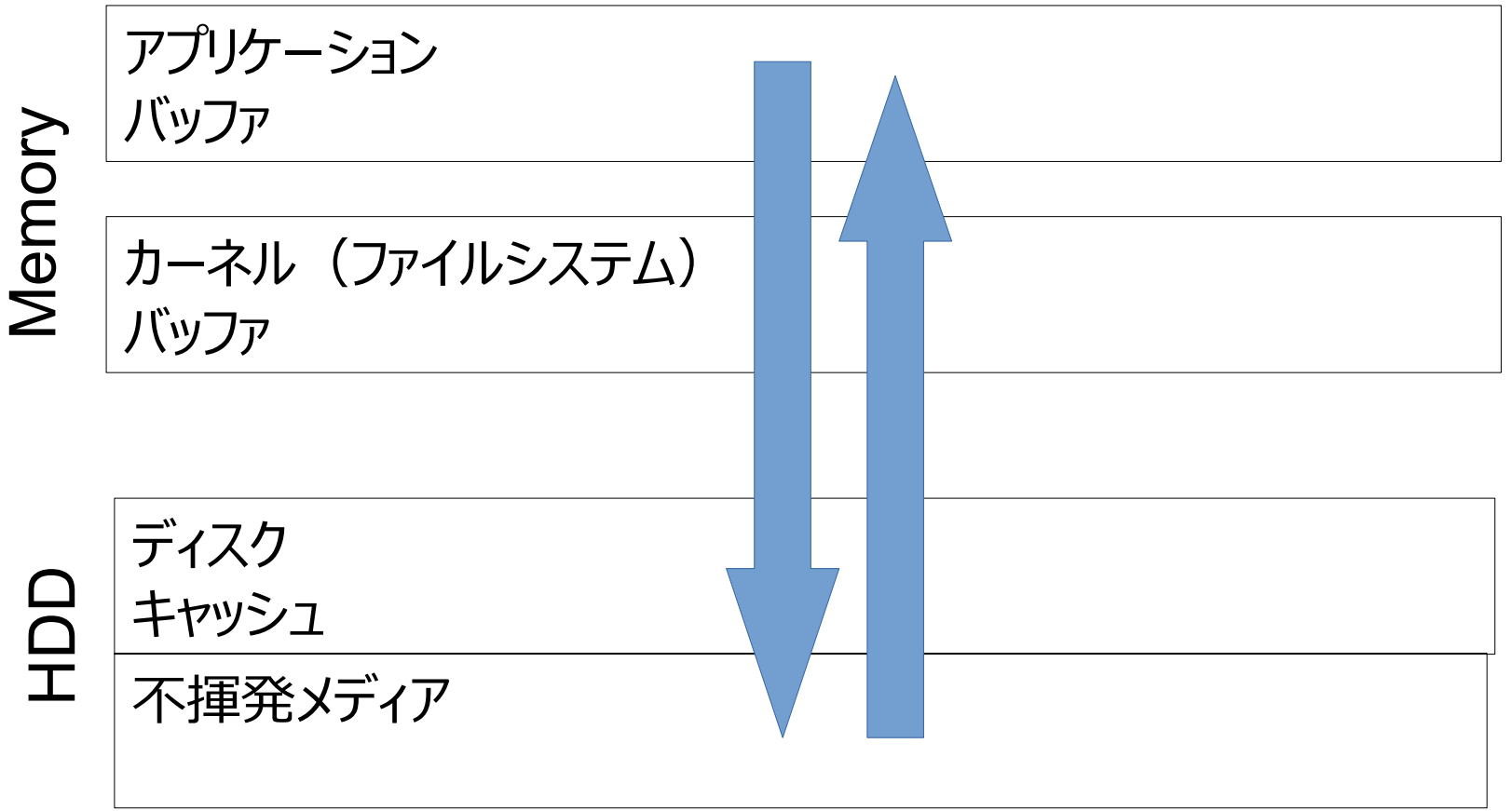
MySQL のメモリ見積り

- 物理メモリ = OS(ntp,cron) 基本部として64M + MySQL分
- MySQL = スレッドバッファ * max_connections + グローバルバッファ + tmp テーブル
- スレッドのバッファ
  sort_buffer_size 256K
  read_buffer_size 128K
  join_buffer_size 256K
  read_rnd_buffer_size 256K
  デフォルト約1MB。実戦では sort_buffer_size と join_buffer_size の増量が必要
- グローバルなバッファ
  key_buffer_size 8M
  innodb_buffer_pool_size 128M ⇒??? （指定より約10%多く取得される）
  innodb_log_buffer_size 8M
  innodb_additional_mem_pool_size 8M
- ソートやJoinで使われる内部のMEMORYテーブル
  - max_heap_table_size = tmp_table_size = 16M (増やす時も同じ値にすること)

デフォルトだと 約255MB + バッファープール +Kernel バッファ分の配慮

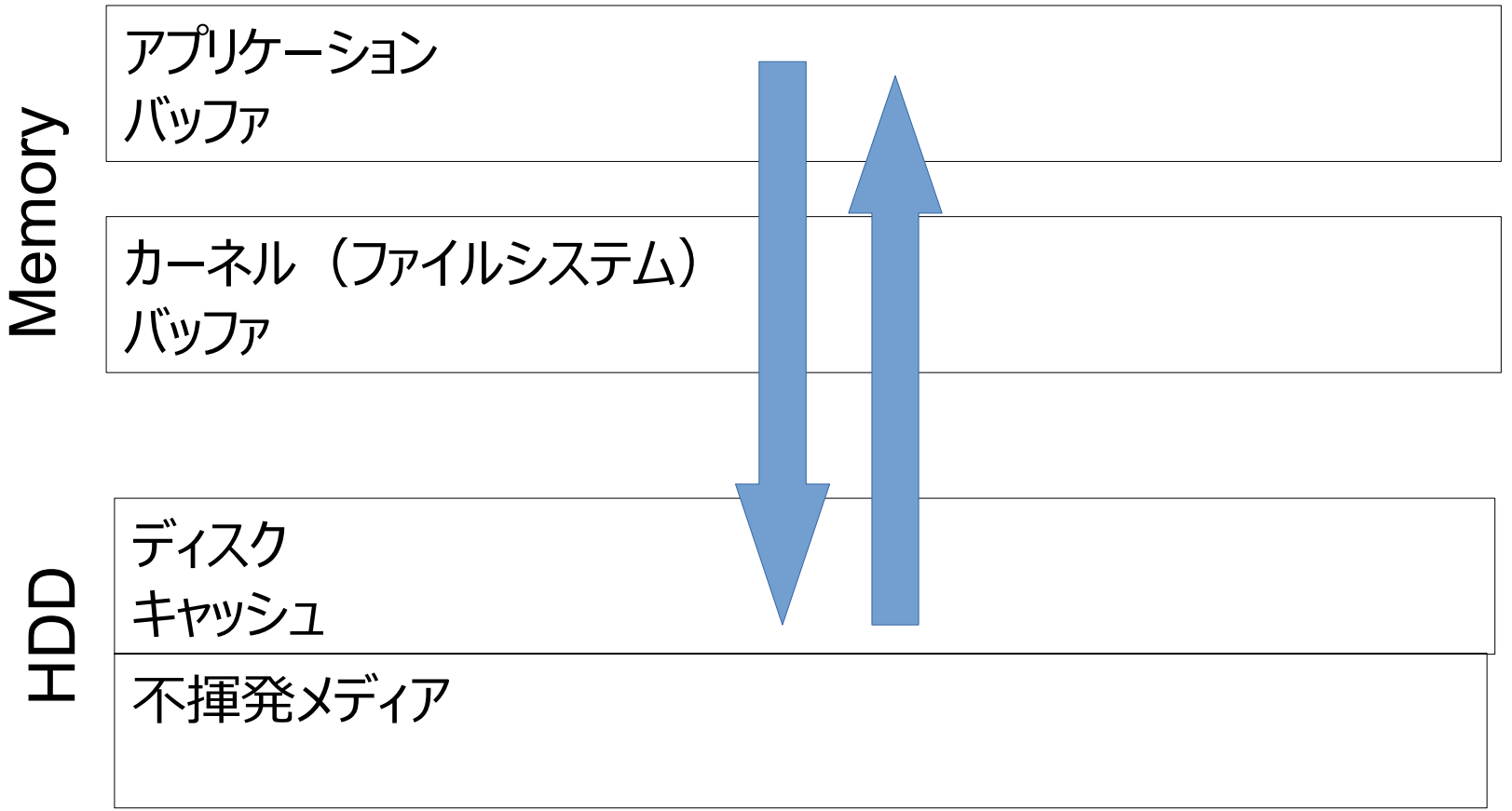
innodb_buffer_pool_size を見積るための長い旅

同期 I/O の O_SYNC をブロックデバイスで



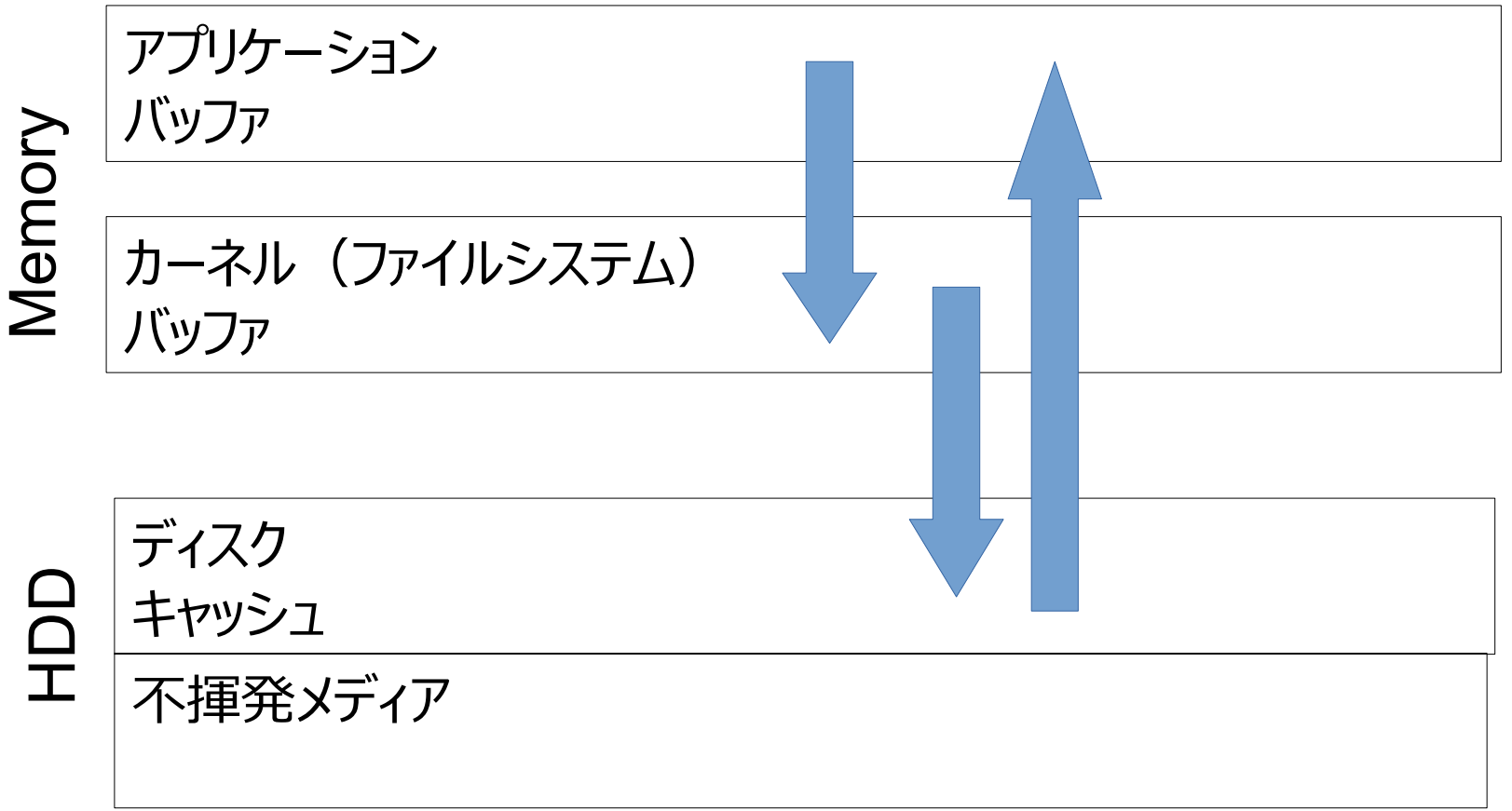
innodb_buffer_pool_size を見積るための長い旅

同期 I/O の O_SYNC をファイルに対して



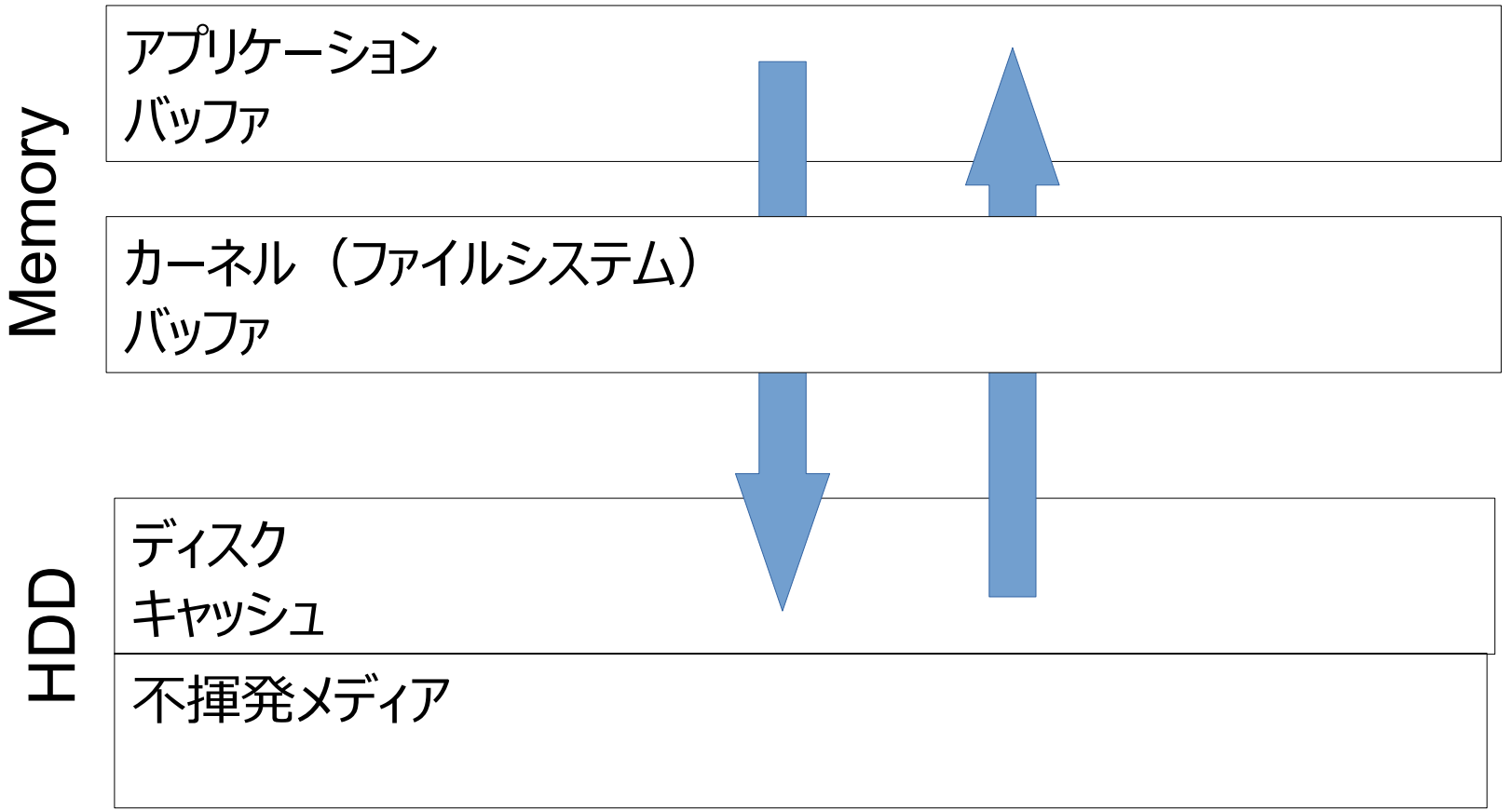
innodb_buffer_pool_size を見積るための長い旅

非同期な write と fsync をファイルに対して



innodb_buffer_pool_size を見積るための長い旅

非同期 O_DIRECT と fsync をファイルに対して



innodb_buffer_pool_size を見積るための長い旅

- MySQL 5.6.7+ での innodb_flush_method 対応表

	データ	ログ	メリット	デメリット
fsync （デフォルト）	fsync	fsync	• カーネルのキャッシュで 2 段キャッシュ • fsyncは汎用なので仮想との親和性が高い	• 二重にキャッシュされ無駄 • カーネルの分考慮してバッファプールやや小さめに
O_DSYNC	fsync	O_SYNC	ログの書き込みで信頼性が高くなる	ログの書き込み速度が落ちる
O_DIRECT	O_DIRECT + fsync	fsync	• バッファプールを目一杯大きく出来る • ホスト直づけの物理ディスクなら速い	• SANだと低速 • ジャーナルFSだとO_DSYNC に変更 • バッファプールに収まらないとディスク直利用で低速

innodb_buffer_pool_size を見積るための長い旅

- 物理メモリの80%をバッファープールで O_DIRECT が鉄板？
  - クラウドではSANの可能性が高く O_DIRECT 低速
  - 実戦では InnoDB のテーブルスペースは、物理メモリを上回る事が多い
- 総合的に判断してデフォルト(fsync)が得策

innodb_buffer_pool_size を見積るための長い旅

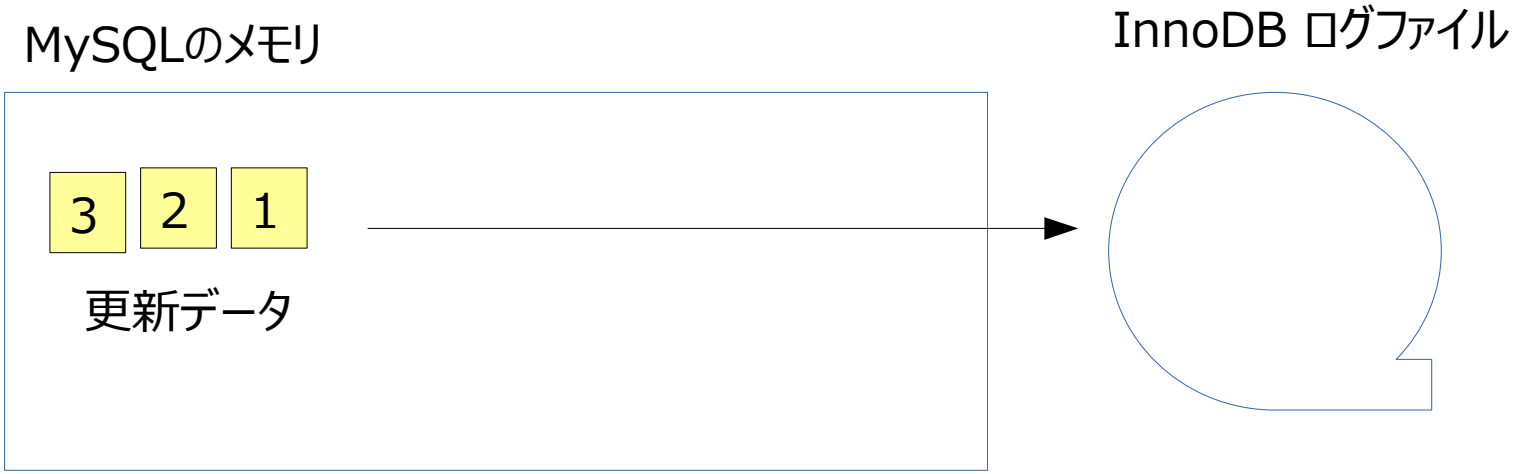
- innodb_flush_method=fsync (default)
- innodb_buffer_pool_size

  ＝ (バッファープールに使ってよいメモリ) ＊ 3/4 で始める
- 例: 1G メモリのマシンでデフォルトのスレッドバッファ設定で

  ((1024M – 255M) ＊ 0.9) ＊ 3/4 = 525M
- バッファープールのサイズが決まったら次はログのサイズです

物理Write量を支配する innodb_log_file_size

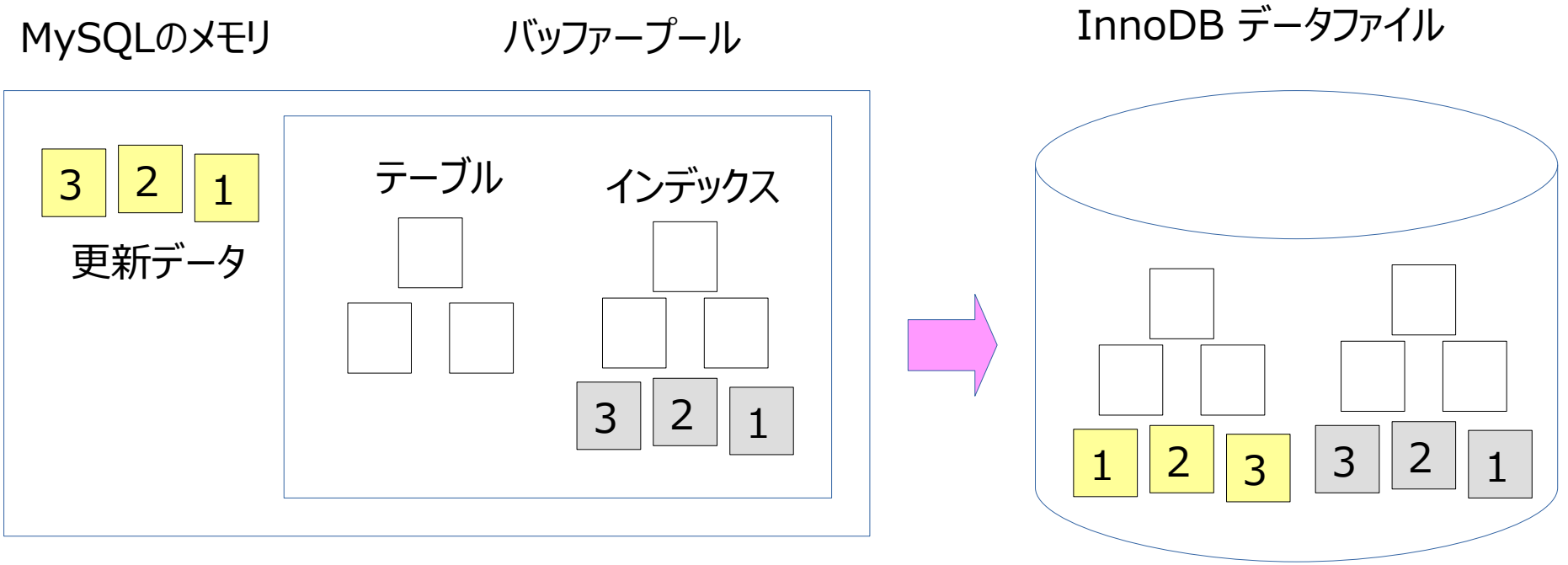
InnoDB ディスクI/O のしくみ　（InnoDB ログファイル）



- 連続データをシリアルにディスクに書き込むので非常に高速
- HDDの SAN で SSD 並に速い
- ログファイルに書く≒更新トランザクション終了
- 最大 4GB (MySQL 5.5), 512GB (MySQL 5.6+)

物理Write量を支配する innodb_log_file_size

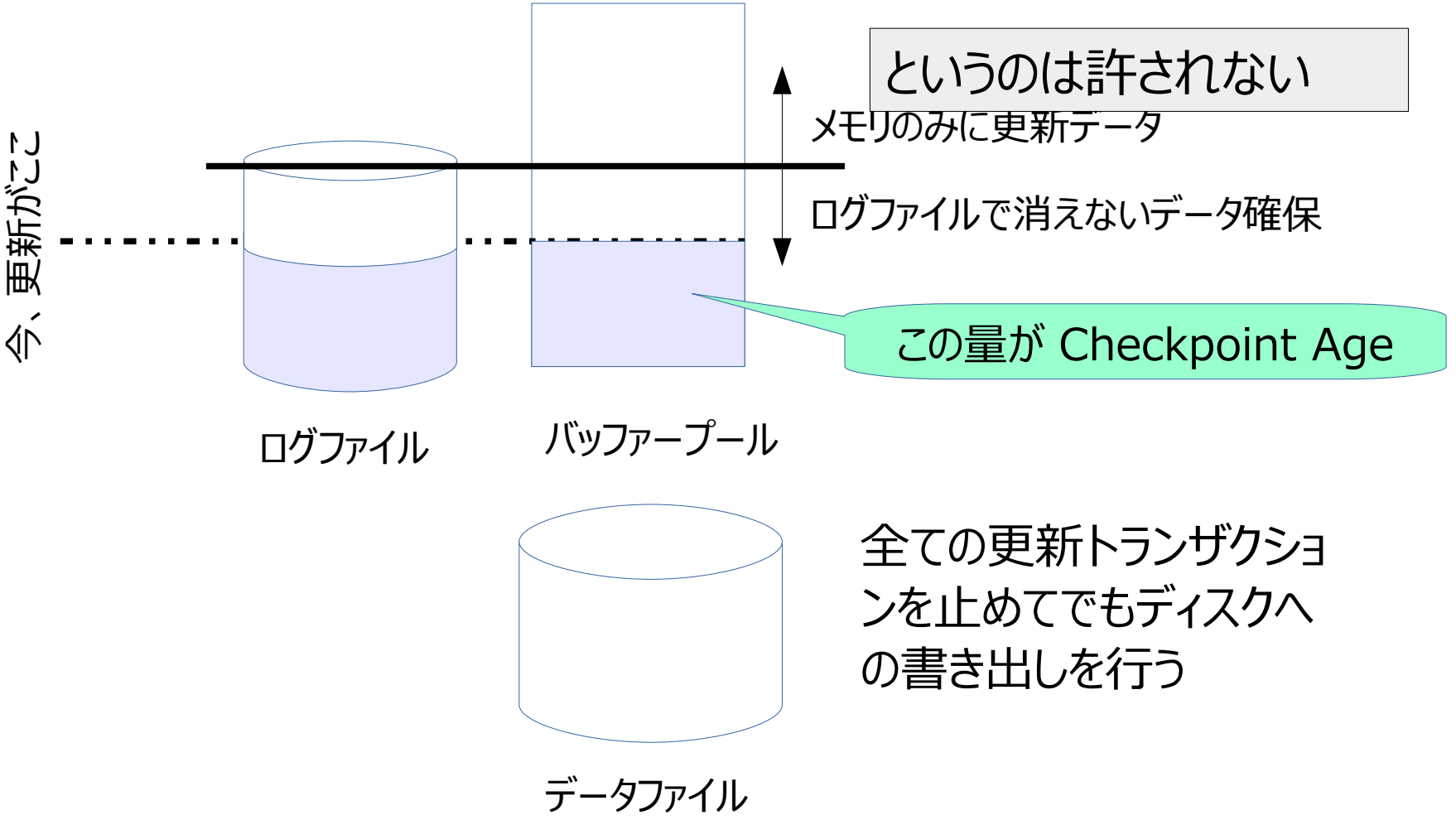
InnoDB ディスクI/O のしくみ　（InnoDB データファイル）



- バッファープールに格納するところまでで更新クエリ終了
- 後続クエリの更新はメモリ上で完結する
- まとめてディスクに書き出す（チェックポイント）
- データ量も多く、ランダムWriteなチェックポイントは重い処理

物理Write量を支配する innodb_log_file_size

InnoDB ディスクI/O のしくみ　（InnoDB データファイル）



物理Write量を支配する innodb_log_file_size

Fuzzy Checkpoint
- アイドル時に定期的に発動しバッファープール上の古いダーティページから1回あたりは少量の書き出しを行う
- 負荷低く影響低いのでチューニング不要

Sharp Checkpoint
- ログファイルサイズの閾値(75～90%)を超えると発動し全てのダーティページを書き出す
- ログファイルサイズが大きいとディスクのWrite量が多くなり高負荷
- Write 中でも更新は入ってくるので、排出（書き出し）速度が負けて使用率100%が続くと新規トランザクションは実行不可に
- バーストI/Oなのでクラウドに合っていて高パフォーマンス出せる場合も

Adaptive Flushing (MySQL5.6+)
- 更新量に応じた書き出しを innodb_io_capacity_max に従って、常時行う
- ディスクの IOPS がプロビジョニングで保証されているなら Sharp Checkpoint の回避に
- IOPSプロビジョニングが無いクラウドでコンスタントに負荷をかけると、遅延の元にも
- デフォルトON

物理Write量を支配する innodb_log_file_size

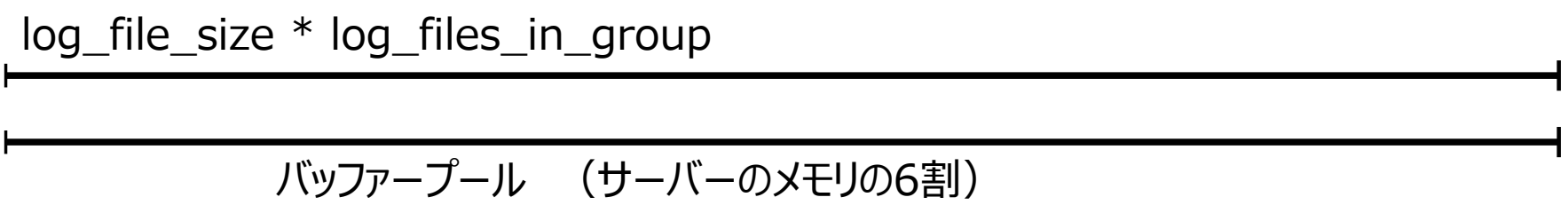
- MySQL 5.6+ の Adaptive Flushing を使うかどうかで MySQL のWriteの性格がガラッと変わる
- デフォルト値
  - innodb_adaptive_flushing=ON
  - innodb_io_capacitiy=200
  - innodb_io_capacity_max=2000
  - innodb_log_file_size=50M, innodb_log_files_in_group=2
  - 100MBのCheckpointが溜まらないよう常時I/Oがかかる
- 1TB SSD で 3000 iops が目安。Adaptive Flushing のデフォルト値はクラウドだと70Gの IOPS プロビジョニングSSD相当（210 iops)
- 物理の場合の参考　1.5K RPM HDD で 200 iops 相当
- クラウドでHDDモデルなら 100 iops/1000 iops 以下に設定

物理Write量を支配する innodb_log_file_size

- クラウドでのディスクI/Oはそもそもバーストなら高性能、コンスタントだと低性能なので、たまった更新量に応じて常時I/Oが発生する Adaptive Flushing を上手く使うのは難しい
- もしSSDモデルなら ⇒ デフォルト設定のまま log_file_size をチューニングしていく
- もしHDDモデルなら ⇒ Adaptive Flushing は OFF にして、log_file_size をチューニングしていく

物理Write量を支配する innodb_log_file_size

ハイリスクハイリターン法：　ログファイルサイズはバッファープールと同等に



利点：

- Adaptive Flushing OFFかつ**１日分の更新が上手くおさまれば**、InnoDBがシーケンシャルなログ書き込みとメモリ上だけで動作出来る

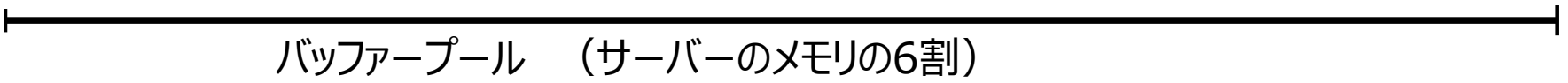
欠点：

- クラッシュ時の処理が長大に
- Adaptive Flushing ON で、プロビジョニングiopsじゃないHDD/SSDだとコンスタントに負荷に
- １日の更新量　の見積りを誤ると、長いトランザクション停止（数分～数時間）が発生して絶大なダメージ

物理Write量を支配する innodb_log_file_size

安全第一法：クラッシュ時の処理も考慮してログファイルサイズほどほど

log_file_size * log_files_in_group　100MB～1GB未満



バッファープール　（サーバーのメモリの6割）

利点：
- クラッシュ時のリカバリも問題なし
- ときどき Sharp Checkpoint が発生するが量が少ないので処理のインパクトが小さい
- **１日の更新量を正確に予測しなくてよい**

欠点：
- ディスクへの書き込みが脈動的に発生しMySQLの性能も脈動的に変化する
- プロビジョニングiops の高性能SSDを使っている場合、性能を使い切れていない可能性

使うか使わないか、それが問題だ Double Write

- InnoDB のテーブルスペースのフラッシュは 16KB のページ単位、一方 Kernel は4KB、ディスクのセクタは 512B 単位なので、OSやハードウェアのクラッシュ時、アプリ側の fsync では書き込み終了なのに、そのページが実際には部分的にしか書かれていないというパーシャルWriteが発生
- 先に Double Write バッファに書いてから、次に同じデータのWriteを行う仕組みが考案された
- HDD内キャッシュ(4MB～)での効果もあり、2 倍はかからず 1.05～1.1 倍程度のオーバーヘッドだった

使うか使わないか、それが問題だ Double Write

- I/Oを高速化する技術
  - プリフェッチ：read 時の連続ブロックの先読み
  - Writeマージ：順序を入れ替えてでも１回の書き込み量をまとめる（これを止めさせるのがWriteバリア）
  - Kernel ⇒ RAID コントローラー ⇒ SSD/HDD のあらゆる段階でこの手法が使われている
- 特定の小領域（Double Write) ばかりを高頻度で書くと、全体のWriteマージやプリフェッチによる高速化に悪影響となる
  - ext4 だと 1.5倍＋に遅延する
  - Double Write でパフォーマンスが低下しないのは物理HDD１本のとき

使うか使わないか、それが問題だ Double Write

- ext4 data=jornal マウントや XFS などジャーナリングFSを使えば Double Write と同様の保護が可能でオーバーヘッドも1.1程度
- クラウドでは積極的にFailoverさせる構成をとっているので、落ちたサーバーは棄てるだけでよい
- Failover が通用しないクラウド全体障害、ゾーン（データーセンター）障害に備えるには、予備サーバーなど特定のサーバーを手厚く保護する
- 全体障害の確率を考えると、支払っているオーバーヘッドに見合わないので、負荷のかかる主力サーバーは、思い切って Double Write は OFF にしましょう

グラフを使っての実践的パフォーマンスチューニング

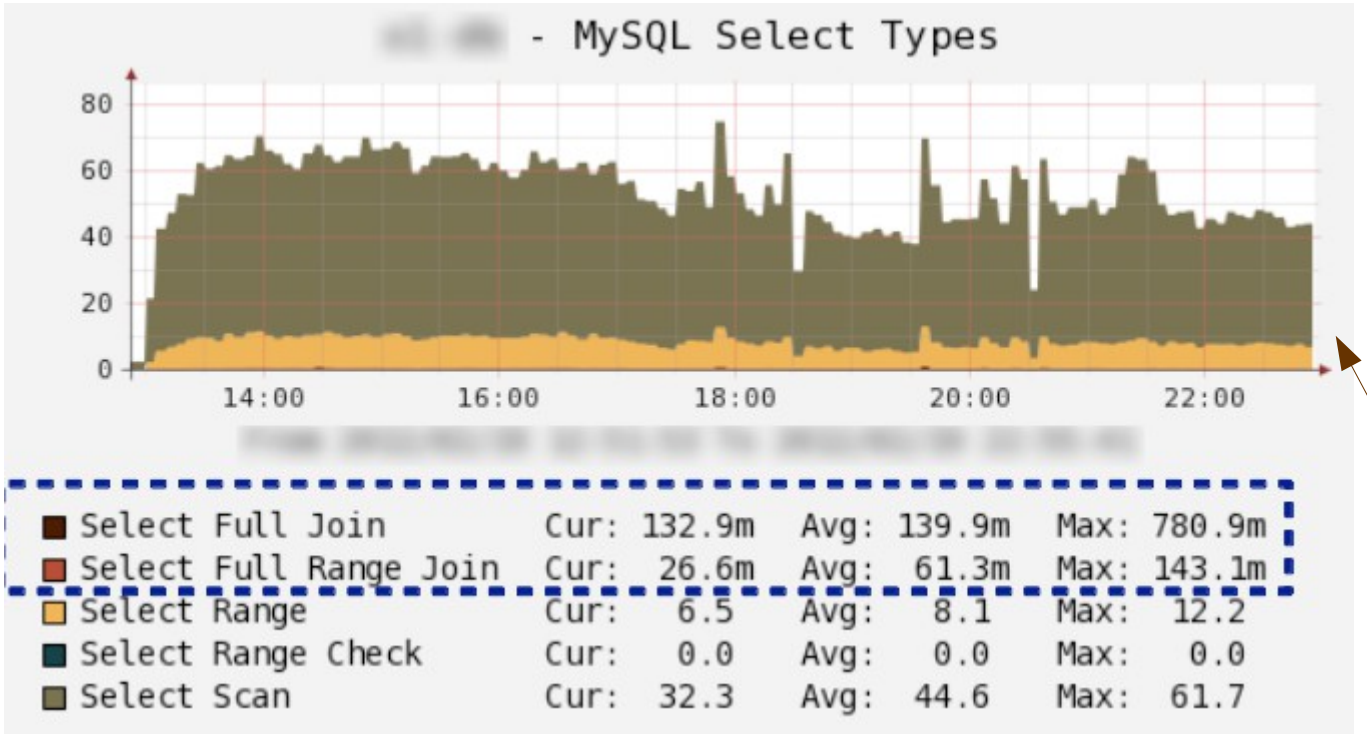
- 負荷試験や本番でのデータを元に絶え間なく改善するのがチューニング
- 威力を発揮するのが、Cacti グラフ

- Percona MySQL Monitoring Template for Cacti
  - Percona 社提供の Cacti のグラフテンプレート
  - MySQL の詳細なグラフ
  - Cacti の Thold プラグインを使えば監視アラートとしてメールを発信、Cacti を監視インフラにすることもできる

Percona のグラフでのチューニング

- クエリのチューニング
  - MySQL Select Types
  - MySQL Handlers
- システム、I/O廻りのチューニング
  - InnoDB Checkpoint Age

MySQL Select Types でアプリのSELECTの書き方を判定

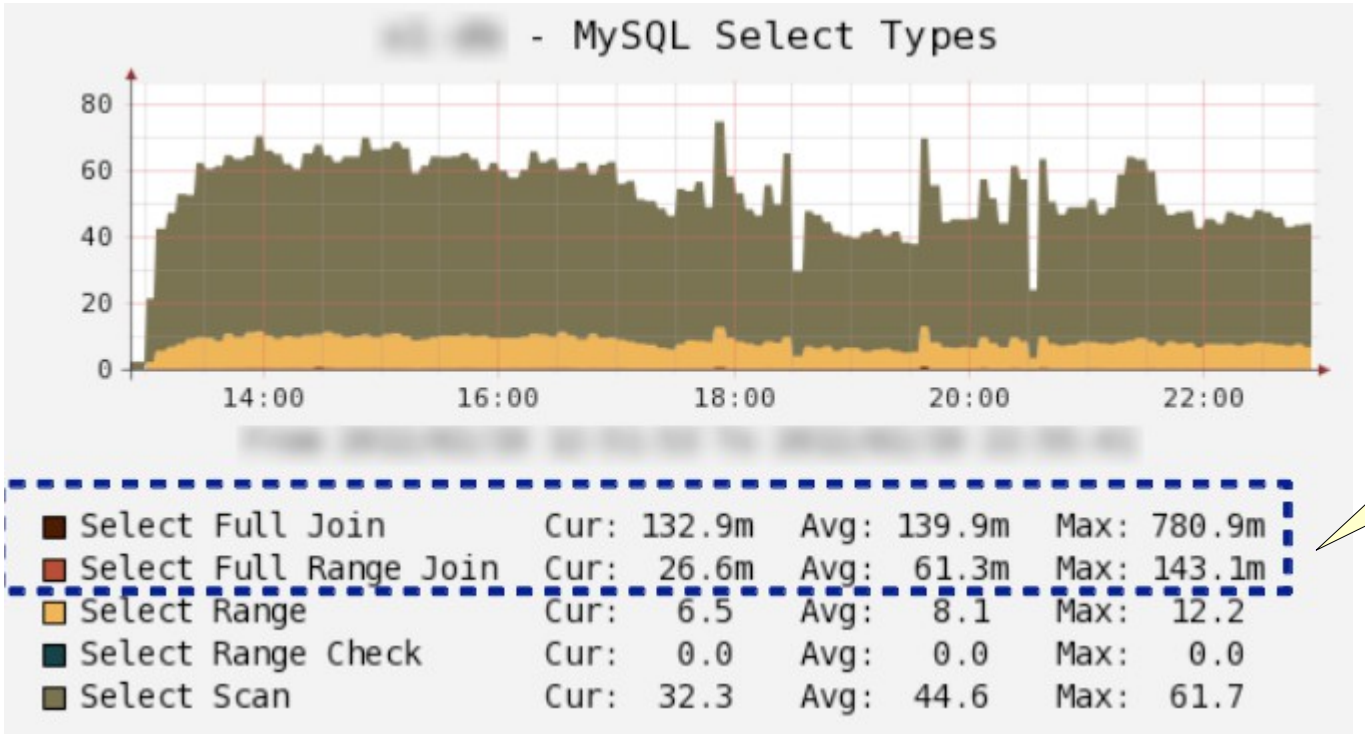


テーブルまたはインデックスで

- Select Scan
  全件検索
- Select Range
  where で範囲を制限してselect

Select Range の割合 >> Select Scan の割合 が望ましい これは NG

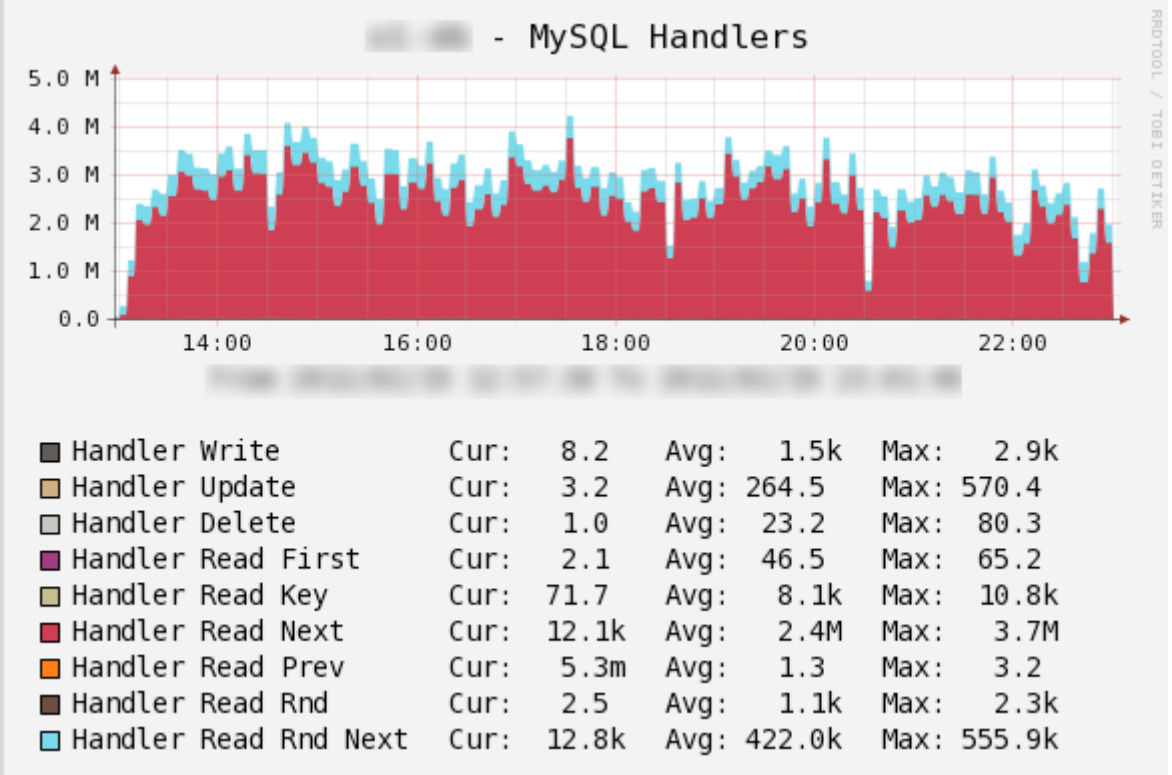
MySQL Select Types でアプリのSELECTの書き方を判定



m(ミリ) なのでグラフでは見えませんが

- 「JOIN するときはインデックスを使って行う」 という鉄則
- 鉄則を破ったクエリが存在

MySQL Handlers グラフでI/O量を把握



MySQL
セッション管理
クエリをHandler命令に
コンパイル

KVS的
Handler 命令

ストレージエンジンデータ
操作

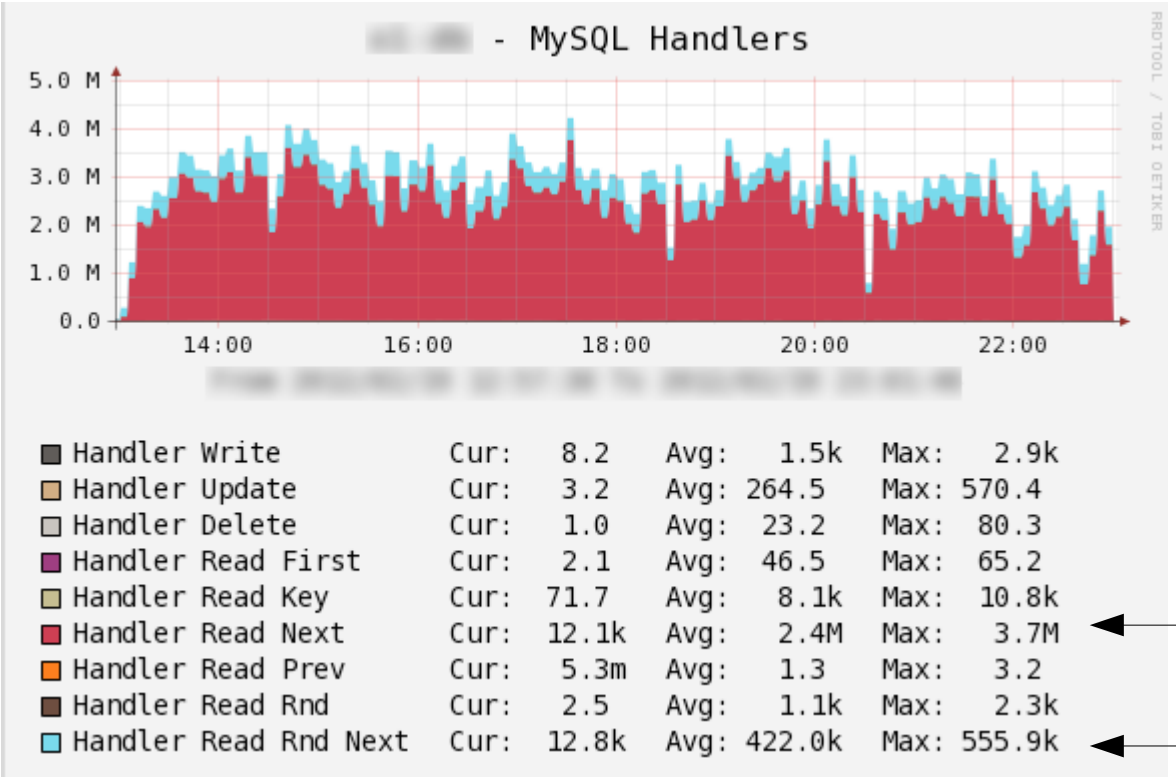
Handler のタイプ詳細

- **Handler Read First** テーブルやインデックスの全件検索スキャンで最初に先頭レコードの取得を行います。その回数。少ないほうが良い
- **Handler Read Key** インデックスのキー値に基づいて行を読んだ回数。
- **Handler Read Next** キー値に基づいて行を特定した後、後続の行を読んだ回数。
- **Handler Read Prev** キー値で行を決めた後、その前の行を取得した回数。
- **Handler Read Rnd** InnoDB でプライマリキーの値を指定して1行読んだ回数。
  ディスクへのアクセス方法がシーケンシャルアクセスではなくランダムアクセスということで、MySQL の世界では歴史的にピンポイントで１行読み込む動作に **Random Read** という用語
- **Handler Read Rnd Next** Read Rnd によって行を読んだ後、後続行を読み取った回数。

グラフの形で観察できます！！！

- **Read Key < Read Next** インデックス使っていても範囲で読み込みしている
- **Read Rnd < Read Rnd Next** Rnd Next の比率が高いと、プライマリキーを使っていても広範に読んでいるのでやはりよくない傾向

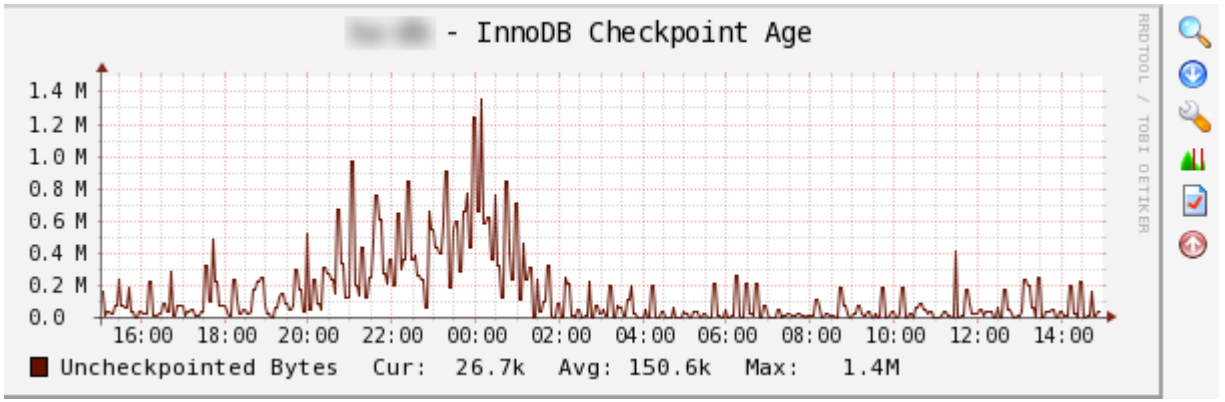
前出の MySQL Handlers グラフだと



Read Next で読み込んだ回数が圧倒的なので、インデックスを使った範囲読み込みの割合が多いことがわかります

システム、I/O廻りのチューニング

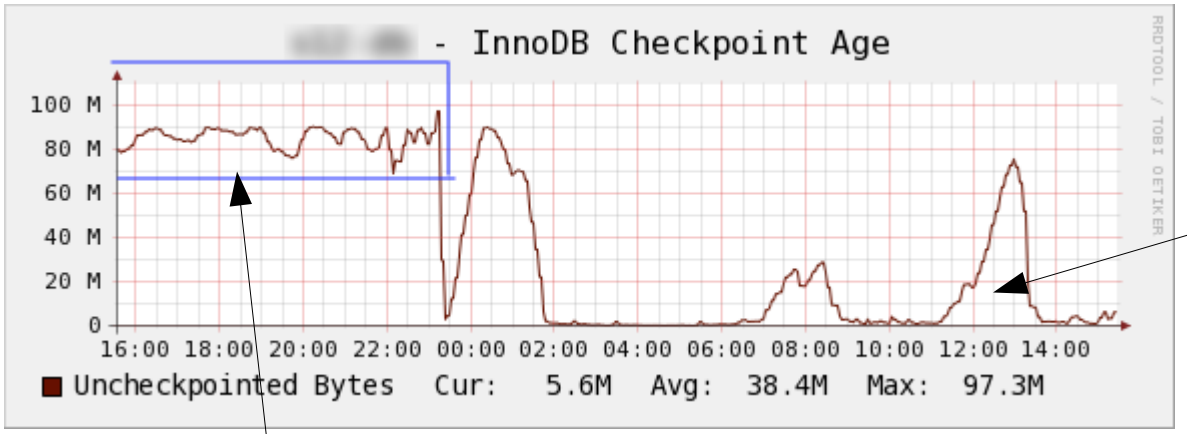
- InnoDB Checkpoint Age



- バッファプール上に更新が溜まる速度に対して、fuzzy checkpoint や adaptive flushing　が打ち勝っている状態
- I/O の能力を使い切っていないのでまだ更新増やせそう

システム、I/O廻りのチューニング

- InnoDB Checkpoint Age

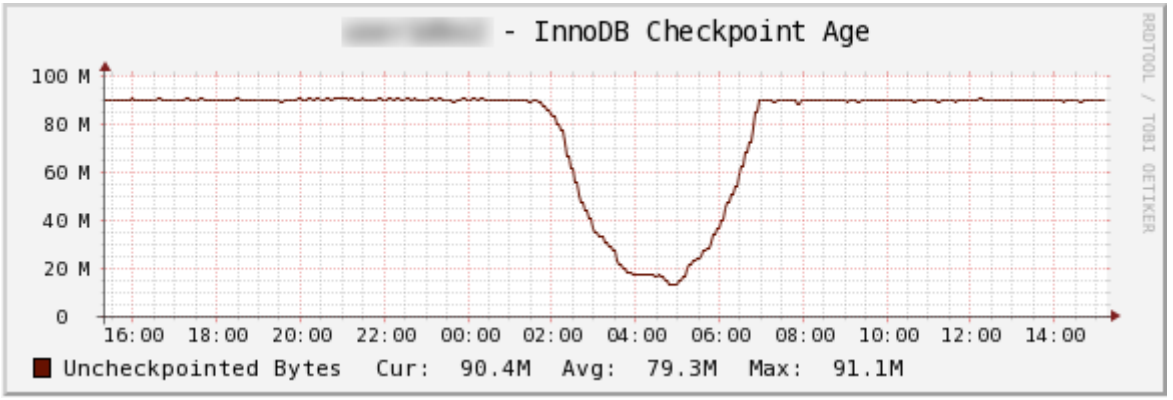


Sharp checkpoint がおこなわれているがすぐ更新をはけているので問題なし

- Sharp Checkpoint が断続的に発生
- Adaptive Flushing がOFFか弱め
- ディスク能力を使い切っているある意味で理想的なバランス
- I/O 負荷による微小な遅延がアプリに影響出ていないか確認しましょう

システム、I/O廻りのチューニング

- InnoDB Checkpoint Age



- 水平線になった場合 sharp checkpoint が途切れることなく走っている状態
- Checkpoint Age 100%到達時のトランザクション停止が微小で済んでいるか、長時間に及んでいるか、グラフでは区別がつきません
- 必ずアプリケーションのログで確認しましょう

まとめ

- 使っているハードウェア、クラウドの環境、実行されているクエリ、サーバーの構成によって最適な設定は異なってきます
- MySQL や Linux のバージョンが変わることでも設定の内容や意味も刻々と変わります
- あなたの MySQL もきっとまだチューニングの余地があります、ノウハウを共有して MySQL を盛り上げましょう！ ！