HBase/Hadoop を利用した J-PARC 運転データアーカイバの改良

IMPROVEMENT OF THE J-PARC OPERATION DATA ARCHIVER USING HBASE/HADOOP

池田浩[#, A, B)], 菊澤信宏[A)], 吉位明伸[C)], 加藤裕子[A)]
Hiroshi Ikeda [#, A, B)], Nobuhiro Kikuzawa[A)], Akinobu Yoshii[B)], Yuko Kato[A)]
[A)] Japan Atomic Energy Agency
[B)] Visible Information Center, Inc.
[C)] NS Solutions Corporation

Abstract

The Linac and the RCS in J-PARC provide enormous operation data to control their equipment, and we have been accumulating the data into a PostgreSQL database system, while we are planning to replace the storage with HBase/Hadoop. HBase is so-call NoSQL, specialized for big-data with scalability to the data size at the cost of the high broad utility of SQL. HBase is constructed on a distributed file system provided by Hadoop, using a cluster with advantages including automatically covering its composing nodes' breakdowns and easily adding new nodes to extend its capacity.

In the previous presentation we reported that we updated the versions of HBase/Hadoop composing our test system, to conquer a single point of failure by making redundant the master node, and we showed issues to fix our tools in the new system, while we also mentioned some issues about the construction of our cluster itself.

In this presentation we are reporting that we have re-designed and re-constructed the cluster with resolving the issues, including enhancing hardware of master nodes, creating scripts to automatically construct nodes, and introducing monitoring tools for nodes. Having accordingly adjusted the configurations of HBase/Hadoop and measured the performance of our new system, we are also reporting its results and considerations.

1. はじめに

J-PARC の Linac 及び RCS では機器から得られるデータを PostgreSQL に蓄積しているが[1]、これを HBase[2]/Hadoop[3]を用いたシステムに変更するための準備を進めている。前回の発表では、テスト用に作られたシステムを運用段階に移行するに問題点をいくつか指摘した[4]。今回の発表ではこれらを含む問題点を解消するべく、新しいクラスタの再設計・再構築について述べ、新クラスタのパフォーマンスについて述べる。

OS は CentOS 6.6 を用いる。なお、Transparent Huge Pages はデフラグ機能に不具合があるため無効にする。Hadoop 2.2.0, HBase 0.96.1.1, ZooKeeper 3.4.5 を用いる。現在ではこれらは既に古くなってしまっているため、ノード構築の手順が一通り確立してから、これらのバージョンを上げることを検討する予定である。

2. クラスタの再設計・再構築

2.1 ハードウェアの増強とメモリの割り当て

前回の発表にて、Hadoop のバージョンアップにより、Hadoop の分散ファイルシステム（HDFS）において単一障害点であったマスタノードの冗長化が可能になった。これにより、DRDB や Heartbeat などの他のソフトウェアの手助けなしに、実用レベルのクラスタを構築できるようになった。ただし、我々が冗長化のために用意したハードウェアは著しくスペックが不足しており、また、RAID も組まれていなかった。このため、ハードウェアを増強し、Hadoop 等のプロセスの配備、及び、メモリの配分を見直した。

Hadoop にてマスタノードを冗長化するに使うノードは最低 3 台必要となる。このうち 2 台は、Hadoop が構築するファイルシステムのメタ情報を記録するモジュールであるネームノードのアクティブとバックアップとなり、同等のハードウェアスペックを有するのが望ましい。3 台目が必要なのは、ZooKeeper 及び Hadoop のモジュールの 1 つであるジャーナルノードが、多数決の原理により決定を行う性質を持つものであるためである。

新しく増強したハードウェアは 2 台でありスペックが高いのでこれをネームノードのアクティブとスタンバイにする。冗長化を行う前に使用していた既存のマスタのハードウェアとあわせて、3 台構成とする。なお、Hadoop の MapReduce 処理を行う基盤となる YARN は冗長化の対象とされていないが、そのマスタは既存のマスタ上で稼動させることとする。各機器のハードウェアスペック及び具体的なモジュールの配備とメモリの割り当てを、それぞれ Table 1 及び Table 2 で示す。

ikedahi@post.j-parc.jp

Table 1: Hardware Specification

Master Node 1	DELL PowerEdge R610 CPU: Intel Xeon E5620 (4Core, 2.4GHz) MEM: 24GB HDD: 600GB x4 (RAID10)
Master Node 2,3	DELL PowerEdge R320 CPU: Intel Xeon E5-1410v2 (4Core, 2.8GHz) MEM: 24GB HDD: 600GB x4 (RAID10)
Slave Node	DELL PowerEdge R410 CPU: Intel Xeon E5620 (4Core, 2.4GHz) MEM: 24GB HDD: 2TB x4

Table 2: Memory Allocation (GB)

Master Node	1	2,3	Slave Node	
ZooKeeper	1	1		
Name Node		8	Data Node	1
ZKFC		1		
Journal Node	1	1		
Resource Manager	1		Node Manager	1
History Server	1		YARN container	6
HBase Master		4	Region Server	12
total	4	15	total	20

2.2 RAID とマウントオプション

Hadoop における実データの冗長化はスレーブノード間で複製を行って実現している。このため、スレーブノードで冗長化のために RAID を組むのは意味をなさない。そもそも、RAID による冗長化はあくまで HDD に対する冗長化であり、ノードそのものを冗長化するわけではなく、代わりにはならない。また、障害発生時のパフォーマンス低下は著しく正常時の数分の 1 に落ちる。このため、ノードそのものに障害が発生したと判断されてクラスタから切り離されたら本末転倒になる（ただし、その方がクラスタとしてのパフォーマンスを維持することができるかもしれない）。RAID コントローラのホットスワップを利用すれば HDD の障害に対する復旧は比較的容易ではあるが、ノード構築の自動化の確立でそれに代えることができよう。

正常時のパフォーマンスについても RAID 0 を使うよりも Hadoop に個々の物理ディスクにアクセスさせた方が速いと言われている。これは、RAID 0 ではもっとも遅いディスクに速度を合わせる必要があるためと説明されている。このため、Hadoop のスレーブノードは RAID を組まないことが常識とされている。ただし、複数のディスクに並列にアクセスする必要がない低負荷の場合は、RAID 0 を組んだ方がパフォーマンス的に有利になるだろう。

前回の発表時では、スレーブノードが物理ディスク 4 台の RAID 5 で構成されていた。RAID 5 はパリティ計算の負荷のために書込みについてパフォーマンスが落ちるとされているが、RAID コントローラがこの負荷を担当することで、物理ディスクが一台少ない RAID 0 構成と同等の性能を発揮する。何れにせよ、障害発生時や高負荷時を考え、Hadoop の常識に従って非 RAID に変更する。正確には、現在使用している RAID コントローラでは非 RAID を構成できないので、1 台構成の RAID 0 を 4 つ構成することとする。RAID コントローラを経由するため、バッテリバックアップ及び不揮発性のメモリを備えた RAID コントローラのキャッシュが利用できる。

OS のファイルシステムとして、現在 Linux で標準的な ext4 を用いる。マウントオプションについてだが、スレーブノードは冗長化しているが最悪のケースを想定し、データの永続化を保証するものとする。RAID コントローラの不揮発性のキャッシュが利用できるため、I/O バリアを無効とする。ジャーナルモードは ordered を用い不正なデータが見える可能性を防ぐ（Hadoop ではランダムアクセスをサポートしていないのでこれで十分となる）。我々の ZooKeeper, Hadoop, HBase では Java 6 をサポートしているが、Java 6 ではディレクトリに対する fsync に相当する命令が用意されていないため、必然的に dirsync が必要となる。

実のところ、Hadoop のスレーブノードでは既定で fsync が実行されず、このままでは上記の設定は意味を成さない。ドキュメントに記述されていないプロパティ dfs.datanode.synconclose を true に設定する必要がある。後述するパフォーマンス測定は、これが判明する前の段階で行ったものであり、旧クラスタ・新クラスタともこのプロパティを設定していない。

2.3 HBase のテーブルの属性とパフォーマンス

HBase のテーブルにおいて、パフォーマンスに影響を与える幾つかの属性について言及しておく。

HBase は、複数のレコードに同じデータが何度も現れることを問題とせず、ファイルに格納する。これは、Data Block Encoding や、Snappy 等の圧縮アルゴリズムを適用することを前提としており、テーブル属性として指定される。Data Block Encoding は HBase 0.94 から導入され、HBase のレコードの構成に沿った自明な圧縮を行うもので、特に行キーが値に比べてかなり大きい場合に有効である。圧縮アルゴリズムはその後に適用される。我々のテーブル設計では Data Block Encoding のロジックが非常に合致する。実際にどのような組み合わせが有効であるかは、後に記述するパフォーマンス測定にて示している。

データ検索については、ブルームフィルタが利用可能である。HBase の実装上、データの検索には複数のファイルを確認する必要があるが、これに掛かるコストを、ブルームフィルタを使って軽減することができる。これはテーブルの属性として設定され、既定では行キーに対してブルームフィルタの検索が可能な設定になっている。ただし、我々の用途では範囲条件による複数レコードの抽出が主たる閲覧方法であり、これにはブルームフィルタを適用できず、ブルームフィルタに要する多少の記録容量がほぼ完全に無駄になってしまうため、ブルームフィルタを無効にする。

さらに、HBase にはミリ秒オーダのレイテンシを実現する目的で Block Cache という機能が用意されている。一度ディスクから読み込んだブロックは次からはメモリ上のキャッシュから参照される。キャッシュ自体はサーバ毎に 1 つ用意されるが、実際に使うかどうかはテーブルの属性で指定する。我々の用途では、テーブルの閲覧では一度に大量のデータを取得し、このためレイテンシの低さ自体はあまり興味がなく、一度読み込んだデータを再度参照することは少ないだろう。また、Block Cache は既定では最大ヒープの 40%まで使う設定となっており、メモリの大量消費と GC の頻発によるパフォーマンスの劣化の可能性も考えられる。このため、Block Cache は無効にする。

2.4 ノードのモニタリング

冗長化しても障害発生時に対応しなかった場合はせいぜい寿命の期待値が 2 倍になるだけであり、障害発生によるサービスの停止やデータの損失は避けられないだろう。特に、マスタスレーブ型のクラスタでは、台数が比較的少ないマスタが弱点となる。Hadoop ではマスタが冗長化されたが、アクティブとスタンバイの 2 台までとなっている。

サーバの状態監視には Nagios という定番のツールを使っている。Nagios では状態異常の検知は全てプラグインとして導入され、Nagios 本体は検知のスケジュールやメールによる異常状態通知を行う。リモートサーバの状態を収集するためには、通常、NRPE というプラグインを使う。Web ブラウザからサーバやサービスの状態の一覧を見ることができるが、この Web アプリケーションも外部モジュールとして Nagios 本体から切り離されている。

通常、Nagios による監視では異常の通知をメールで受け取るが、我々のクラスタが配置される制御 LAN は基本的に外部のネットワークと切り離されており、制御 LAN 内部からメールサーバへの通信が遮断されている。このため、SSH 経由でログインし、Nagios が Web アプリケーション用に更新するファイルを定期的にパースして状態を確認する。あるいはそれを自動的に行うスクリプトを走らせるものとする。なお、このスクリプト自身が異常終了した場合は目で見て判断できるが、将来的にメールでノードの異常を通知するようになった場合は、Nagios 本体の異常終了に対する対策を考えておく必要がある。

Nagios のプラグインは、サーバに対する ping など定番のものは予め幾つか用意されているが、それ以外は必要に応じてスクリプトを書いたり、信頼のおけるサイトから有用なものを別途ダウンロードしたりする必要がある。我々のクラスタでは、ハードウェアの状態監視に対しては、OS のインストールディスクに含まれるソフトウェア freeipmi を利用して IPMI 経由でハードウェア状態を取得するプラグイン check_ipmi_sensor と、SNMP 経由で各機器にインストールされた Dell の OpenManage にアクセスしてハードウェア情報を取得する check_openmanage を使う。ソフトウェア（ZooKeeper, Hadoop, HBase の各モジュール）の状態監視については、JDK に含まれる jps コマンドを用いてプロセスの存在確認をし、また、ソケット通信にて問い合わせできるものに関してはその応答を確認するプラグインを自作してこれを使う。これは、手動での確認を自動化するということを念頭に置くものである。

Nagios の他には旧クラスタにインストールされていた Ganglia をインストールしている。これは Nagios と異なり数値情報を収集するもので、CPU やリソースの異常消費等、不具合や状態異常のアナログ的判断や、パフォーマンスチューニングに適しているだろう。マルチキャストを使うため、ネットワークの負荷が軽いという特徴がある。

Nagios も Ganglia も JMX 経由で Hadoop や HBase から内部情報を取得できる。しかし、未ドキュメントのパラメータが多いため、それが指し示す意味と妥当な閾値を決めることは、ソースコードを解読するなど熟練が必要だろう。

2.5 ノード構築の半自動化

あらかじめ作成されたノードから、どのような設定、チューニングが行われたかを確認するのはほぼ不可能である。手動でノードを構築するとしてその手順が漏れなくドキュメント化されていれば障害発生時に元の状態に戻すことはできるが、ドキュメントへの記入漏れやノード構築時のヒューマンエラーなどのリスクは避けられない。また、それらを正しく遂行したとしても、手動でのノード構築には時間が掛かる。特にマスタノードに対する冗長性の損失は、早急に回復する必要がある。一方、どのような設定を行ったかを客観性・曖昧さを排除し、再現性・再評価可能である形で示すことは、意味のある評価を行うのに重要である。

ノードの構築ではその手続きを全て自動化するのが望ましいが、SSH の鍵の設定や IPMI 経由で取得するハードウェア状態に関する設定などは、セキュリティ的な観点及び環境依存が強いものは手動で行うこととする。

ノードの構築は OS のキックスタートから始める。OS のインストールディスク（DVD）を用い、キックスタートファイルは USB メモリ経由で与える。インストールディスクから OS をインストールし、OS を稼動させるのに必要とされるソフトウェアの他、ディスクに含まれるソフトウェアモジュールで我々のクラスタが必要とするものをインストールする。OS やこれらのソフトウェアの設定ファイルもここで編集する。OS のインストール後の再起動後、SSH 鍵の手動設定を行い、SSH 経由で、残りのソフトウェアのインストール及び設定を行うためのセットアップ一式をコピー及び実行を行う。

なお、ここでは、ハードウェア構成をおよそマスタノードおよびスレーブノードで単純化しており、将来的にさまざまなハードウェア構成のノードをクラスタに追加していく場合は、このままのキックスタートでは対応できずに、新しい対応をする必要があるだろう。しかし、手始めとしては悪くないだろう。

3. パフォーマンス測定

汎用的な測定ではなく、J-PARC における実際の使われ方を想定したデータの登録・取得のパフォーマンスを測定する。クラスタを再構築する前の旧クラスタについても測定を行った。

3.1 測定条件

データ登録の測定では、EPICS レコード 10000 個から 1 秒周期でデータを取得して格納することを想定し、1 日分のデータがどの程度の所要時間でクラスタに登録できるかを、継続して 5 日分測定する。1 つ 1 つのレコードは非常に粒度が小さいため、実際の運用では、バッファにまとめてから送信するだろう。これに従い、auto-flush は off にし、最後に flush を行い、これが完了するまでを測定する。EPICS のチャネル名として 30 文字を想定し、大文字・小文字のアルファベット計 52 文字から構成される文字列をランダムで 30 文字の名前を生成し、データは 0.0 以上 1.0 未満のランダムな倍精度実数を使う。理想的には、テーブルの作成時に予めリージョンを適切に分割しておくのが望ましい。実際の運用では、どのように適切に分割するべきかは自明でないが、バランサが適時働いて徐々に負荷がスレーブノード間で分散していくことは期待できよう。このツールでは短い時間でパフォーマンスを測定するため、予めリージョンを分けてテーブルを作成し、初めから負荷分散をする。具体的には、ランダムに生成した名前が一様分布しているとし、アルファベット 1 文字を使ってできるだけ均等にリージョンを分割して、リージョンサーバ数と同じだけのリージョンを作る。

データ取得の測定では、先に登録したデータを利用する。5 個のクライアントが同時に接続していることをエミュレートして 5 個のスレッドを用い、それぞれ、10 個のランダムなチャネルに対し、ランダムな日付の 1 日分のデータを取得させ、これが完了するまでの時間を測定する。もともと HBase では、Hadoop のスループット重視とは対照にレスポンスを重視しており、また、同時アクセスする複数のクライアントへの対応を複数のリージョンサーバに分散することで、総合的なパフォーマンスの向上が期待できるものである。しかし、実際の我々の使い方では、それほど多くのクライアントが同時接続することを想定しておらず、また、単体のクライアントに対する大量のデータ取得のスループットに興味がある。

HBase のクライアントにおける書込みバッファは既定で 2MB となっておりこれを用いる。また、HBase におけるデータの取得では 20000 レコードのキャッシュを用いる。これは 1 レコードを 100bytes と換算しておよそ 2MB となる。

先に述べたように、HBase のテーブルの属性として、Data Block Encoding と圧縮アルゴリズムとして何を組み合わせればよいかは不明瞭であり、これらの組み合わせについて測定を行った。具体的には、前者は｛none(適用なし), diff, fast_diff｝について、後者は｛none(適用なし), GZ, Snappy, LZ4｝について測定した。

3.2 測定結果

測定結果をまとめたものを以下の表に示す。Table 3 はデータ登録の所要時間で、1 日毎に得られるデータを 5 日分で平均したものである。Table 4 はこのうち none-none のケースについて、データ登録の所要時間の日推移を示したものである。Table 5 はデータ取得の所要時間で、スレッド毎に得られるデータを平均したものである。Table 6 は Hadoop の分散ファイルシステム（HDFS）上に書き込まれたデータの総量を示す。正確にはこれは新クラスタに対するデータだが、これらのデータはおよそ Data Block Encoding と圧縮アルゴリズムに依存する。

Table 3: Write Test (min)

	none	GZ	Snappy	LZ4
old cluster				
none	102.3	59.0	64.3	64.8
diff	57.4	54.0	58.9	59.4
fast_diff	56.7	55.9	58.5	58.8
new cluster				
none	81.9	77.6	74.9	80.8
diff	67.6	68.6	75.4	87.3
fast_diff	69.0	71.7	76.4	71.4

Table 4: Write Test Details of the none-none Case (min)

day	old cluster	new cluster
0	75.6	73.8
1	91.5	80.2
2	108.2	83.1
3	98.6	86.3
4	137.6	86.3

Table 5: Read Test (sec)

	none	GZ	Snappy	LZ4
old cluster				
none	7.3	8.3	8.8	7.1
diff	8.9	10.0	11.0	9.4
fast_diff	7.5	8.9	8.2	9.4
new cluster				
none	7.2	8.2	8.0	8.5
diff	9.8	8.5	10.0	11.3
fast_diff	8.6	8.6	9.0	8.3

Table 6: Sum of Stored Files (GB)

	none	GZ	Snappy	LZ4
none	286.2	64.1	80.5	76.7
diff	53.1	47.3	52.9	53.1
fast_diff	53.8	48.7	53.8	53.9

3.3 考察

書き込みに関しては旧クラスタが 20～40%ほど速い。ただし、HDFS に書き込むデータ量が極端に多い none-none のケースでは、新クラスタの方が速く、旧クラスタは急速にパフォーマンスが劣化している。これは、負荷が低い場合は旧クラスタにおける RAID コントローラの支援を受けた RAID 5 の速度が効くが、負荷が高い場合は新クラスタにおけるディスクへの並列アクセスの効果が高いためと考えられる。新クラスタの優位性を示すには、テストで使用したデータ量が圧倒的に足りなかったようだ。

読み込みに関しては、旧クラスタと新クラスタで大差がない。ここでは示していないが、テストに用いたチャネル数が少ない場合や再読み込みで OS のキャッシュや HBase の Block Cache を利用する場合は、旧クラスタが速い傾向がある。おそらくこれも、ディスクへのアクセスが少なくなる場合は、相対的に RAID 5 の効果が高くなるためと考えられる。

Data Block Encoding と圧縮アルゴリズムについては、我々のテーブル設計では Data Block Encoding として diff や fast_diff を使うのが圧縮の効果が高く、また書込み速度も高い。テストで使用したデータでは常にデータが変化するため diff の圧縮率が僅かに高いが、実際の J-PARC での運用では変動の無いデータも記録するため、fast_diff の効果が高いだろう。diff や fast_diff を適用した場合、圧縮アルゴリズムとして圧縮率の低い Snappy や LZ4 を適用しても圧縮の効果が現れず、変換に要する時間だけ無駄のようだ。圧縮率の高い GZ はその変換速度は Snappy や LZ4 の数分の 1 とされているが、目立った速度の低下は見られない。おそらく、ハードウェアのスペック上、I/O に比べて CPU の能力が高いためと思われる。

まとめると、新クラスタが旧クラスタよりもデータ量に対するスケーラビリティを持つことが示唆されるが、明らかに示すにはテストに使ったデータ量が圧倒的に足りなかった。Data Block Encoding の diff や fast_diff は効果が高く、さらに圧縮アルゴリズムを適用するなら GZ を使うのが良いことが示された。

4. 今後の対応

4.1 設定の修正とその影響の確認

前述の通り、Hadoop のスレーブノードでは既定で fsync が実行されない。OS のファイルシステム ext4 には遅延アロケーションが導入されており、最大 30 秒のデータ損失の可能性がある。fsync を実行させるには、ドキュメントに記述されていないプロパティ dfs.datanode.synconclose を true に設定する必要がある。Hadoop の設定ファイルの修正及びノード構築スクリプトの修正を行い、パフォーマンスへの影響を確認する。ただし、I/O バリアを無効化しているので RAID コントローラのキャッシュへの掃出しまでに留まるため、ほぼ影響を受けないものと思われる。

このようなドキュメント化されていない重要なプロパティについては、一度、ソースコードを注意深く解読するか、Hadoop や HBase のバグトラッキングに使われている JIRA を注意深く追跡する必要があるだろう。

4.2 HBase 等のバージョンの更新

HBase や Hadoop、ZooKeeper のバージョンを更新することを検討する。これまでにこれらのバージョン互換性の問題に巻き込まれたことがあったため、先ず運用レベルのクラスタを準備することについてフォーカスを当ててきた。これを達成したことを受け、HBase 等のバージョンの更新とそれに伴う調整・修正等を行う。

4.3 スクリプトの汎用化

現在、ノード構築に使われる OS のキックスタートファイルやセットアップファイルでは、簡単のため、IP アドレスなどをハードコーディングしている。このため、別の環境で利用することが難しい。何らかの形でこれに対応しておきたい。

4.4 ノードの増設手順のドキュメント化

ノードの交換手順についてはドキュメント化したが、ノードの増設手順に関してもドキュメント化しておく必要があるだろう。

4.5 データ収集ツールの冗長化

データストアを冗長化したが、データ収集ツールが冗長化していないと、データ収集に関して単一障害点となる。ZooKeeper が稼働しているので、これを用いてアクティブ・スタンバイを決定するのが素直だろう。

4.6 EPICS レコードからのデータ収集以外に適用

現在、EPICS レコードからのデータ収集にフォーカスを当てているが、その他、大量データ収集が必要な状況への適応を検討する。

参考文献

[1] S. Fukuta et al., “Development Status of Database for J-PARC RCS Control System (1)”, Proceedings of the 4th Annual Meeting of Particle Accelerator Society of Japan, August 2007.
[2] http://hbase.apache.org/
[3] http://hadoop.apache.org/
[4] N. Kikuzawa et al., “DEVELOPMENT OF TOOLS FOR THE J-PARC OPERATION DATA ARCHIVING USING HBASE/HADOOP”, Proceedings of the 11th Annual Meeting of Particle Accelerator Society of Japan, August 2014