Management of large amounts of data is a challenge that provides the opportunity to explore different approaches to managing complex states and analyzing numerous metrics. Our professional golf and tennis tournaments generate terabytes of data from site traffic alone. The Predictive Cloud Computing (PCC) system utilizes IBM DB2 to store aggregated information generated from the source data and Graphite to analyze metrics and profile our codebase. Each of these tools gives the PCC system the ability to store and analyze large amounts of data and provide straightforward retrieval for latent analysis.

In this article, we describe data storage in the PCC system, using IBM DB2 as a data source, in conjunction with the Java™ Persistence API. In addition, we discuss how we used Graphite to instrument our codebase and workloads. Finally, we describe the tools we used to analyze that data.

To view this video, Liquibase, please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

**IBM DB2 and the Java Persistence API**

DB2 is an IBM relational database server that was used extensively throughout the PCC system as a persistent data store. Our primary interface to and from DB2 was through Java's Persistence API (JPA2). The Java Persistence API provides a way to map Java objects to relational data, such as rows in tables or views in a database. We also used Liquibase to update the DB2 database schemas, as it provided schema versioning and rollback capabilities. Schema updates were described in an XML markup, and each schema change was part of an individual change set entry.
This allowed for smaller, quicker changes to be rolled out in an individual fashion to the database schema.

“The Predictive Cloud Computing system utilizes IBM DB2 to store aggregated information generated from the source data and Graphite to analyze metrics and profile our codebase. Each of these tools gives the PCC system the ability to store, analyze, and retrieve large amounts of data.”

The PCC system used the Apache OpenJPA implementation of JPA2; to make a connection to a data source with JPA2, the data source is described in a persistence.xml file. An example persistence.xml file is shown in Listing 1.

**Listing 1. The OpenJPA persistence.xml configuration file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
    <persistence-unit name="Aviator-Unit" transaction-type="RESOURCE_LOCAL">
        <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>
        <class>com.ibm.ei.persistence.jpa.CloudStatisticsDAO</class>
        <class>com.ibm.ei.persistence.jpa.CrawlerChecksum</class>
        <class>com.ibm.ei.persistence.jpa.EventPredictionCountDAO</class>
        <class>com.ibm.ei.persistence.jpa.EventStatisticsDAO</class>
        <class>com.ibm.ei.persistence.jpa.LogCount</class>
        <class>com.ibm.ei.persistence.jpa.HistoricalLogCount</class>
        <class>com.ibm.ei.persistence.jpa.Path</class>
        <class>com.ibm.ei.persistence.jpa.PlayerContentAnalysisDAO</class>
        <class>com.ibm.ei.persistence.jpa.PlayerDAO</class>
        <class>com.ibm.ei.persistence.jpa.CrawlerPlayerPopularity</class>
        <class>com.ibm.ei.persistence.jpa.SiteDAO</class>
        <class>com.ibm.ei.persistence.jpa.golf.FeaturedGroupDAO</class>
        <class>com.ibm.ei.persistence.jpa.golf.HoleDAO</class>
        <class>com.ibm.ei.persistence.jpa.golf.RoundDAO</class>
        <class>com.ibm.ei.persistence.jpa.tennis.Match</class>
        <class>com.ibm.ei.persistence.jpa.tennis.MatchStatus</class>
        <class>com.ibm.ei.persistence.jpa.tennis.TennisCourt</class>
        <class>com.ibm.ei.persistence.jpa.twitter.Mention</class>
        <class>com.ibm.ei.persistence.jpa.twitter.PlayerSummary</class>
        <class>com.ibm.ei.persistence.jpa.twitter.Retweet</class>
        <class>com.ibm.ei.persistence.jpa.twitter.TweetDAO</class>
        <class>com.ibm.ei.persistence.jpa.twitter.TwitterUser</class>
        <class>com.ibm.ei.persistence.jpa.HistoricalLogCount</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>
        <properties>
            <property name="openjpa.DynamicEnhancementAgent" value="true"/>
            <property name="openjpa.RuntimeUnenhancedClasses" value="unsupported"/>
            <property name="openjpa.ConnectionDriverName" value="org.h2.Driver"/>
            <property name="openjpa.ConnectionURL" value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
            <property name="openjpa.jdbc.Schema" value="eiblueus"/>
            <property name="openjpa.DataCache" value="false"/>
            <property name="openjpa.QueryCache" value="false"/>
            <property name="openjpa.RemoteCommitProvider" value="sjvm"/>
        </properties>
    </persistence-unit>
</persistence>
```
In persistence.xml, each persistence-unit describes a JPA data source. The provider contains the class name of the JPA implementation, and acts as an entry point for the initialization of the data source. Listed after the provider are the classes that should be enhanced by the OpenJPA compilation phase. These classes are the Java objects that map to relational data; they are explored in further detail later in this article. Finally, any custom configuration properties are specified in the final properties element. Details such as the connection URL, database schema name, and caching configuration are configurable in the properties. The persistence.xml file should be packaged into the Java jar META-INF directory to be readable by JPA at runtime.

After the JPA data source has been configured, classes listed in the persistence.xml can be enhanced and used for accessing database data. Listing 2 shows PlayerDAO, a class that contains fields for information on a golf or tennis player. The methods in the class have been omitted for brevity.

Listing 2. The PlayerDAO class provides access to the PLAYERS table in the database.

```java
@Entity
@Table(name = "PLAYERS")
@IdClass(PlayerId.class)
public class PlayerDAO implements Serializable, Player {

  @Id
  @Index
  @Column(name = "ID", length = 20)
  private String playerId;

  @Id
  @ManyToOne
  private SiteDAO site;

  @ElementCollection
  @Enumerated(EnumType.STRING)
  @Column(name = "PLAYER_TYPES")
  private Set<PlayerType> playerTypes;

  @Column(name = "FIRST_NAME")
  private String firstName;

  @Column(name = "LAST_NAME")
  private String lastName;

  @Column(name = "SEED", nullable = true)
  private Short seed;

  @Column(name = "RANK", nullable = true)
  private Short rank;

  @Column(name = "GENDER", length = 1)
  @Enumerated(EnumType.STRING)
  private Gender gender;
```
There are a number of annotations on and in this class. Annotations are the primary means of telling the JPA enhancer details about the object and what database information or schema the objects map. At the top, @Entity tells OpenJPA the class represents a database relationship, and @Table(name = "PLAYERS") adds information about the table the information should be retrieved from. Therefore, each PlayerDAO object represents a row in the PLAYERS table. The @IdClass(PlayerId.class) annotation tells OpenJPA where to find the class that represents the table’s composite key. A composite primary key contains multiple fields from a table in a database to check uniqueness of a row. We used a composite key for players, so that each player in a tournament would be a separate row in the PLAYERS table.

According to the Java Persistence API specifications, each entity has to implement Serializable interface, and ours also implemented Player, which provide visibility to a number of getter and setter methods. The fields in PlayerDAO are also annotated, with the @Id annotation identifying the fields that make up part of the composite key for the class. The @Column annotation maps the column in the table to the field in the object. There are also a few other annotations, such as @Enumerated, which maps a string in the database table to a Java Enum type, and @ElementCollection, which maps the field to a one-to-many relationship with another table.

Listing 3. The PlayerID class for the PlayerDAO entity object in Listing 2

```java
public class PlayerId {
    SiteId site;
    String playerId;

    public PlayerId(final String playerId, final SiteId site) {
        this.playerId = playerId;
        this.site = site;
    }

    public PlayerId() {
    }

    @Override
    public boolean equals(final Object obj) {
        if (obj == this) {
            return true;
        }
        if (obj instanceof PlayerId) {
            final PlayerId other = (PlayerId) obj;
            return Objects.equal(playerId, other.playerId) && Objects.equal(site, other.site);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hashCode(playerId, site);
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this).add("Player ID", playerId).add("Site ID", site).toString();
    }
}
```
In Listing 3, the PlayerID class for the previously mentioned PlayerDAO class is shown. The class is a simple "Plain Old Java Object" (POJO), with two constructors, and the equals, hashcode, and toString methods. At a minimum, any composite key class needs to share fields that map to the @Id fields in the previous object and an empty constructor. The other methods are implemented to follow Java best practices.

Listing 4. How to use the Java Persistence API to query for all 2016 Australian Open players

```java
// load out persistence unit to the data source
EntityManagerFactory emf = Persistence.createEntityManagerFactory("Aviator-Unit");

//create a connection to the database
EntityManager manager = emf.createEntityManager();

//load all the players for this tournament
final CriteriaBuilder criteriaBuilder = manager.getCriteriaBuilder();
final CriteriaQuery<PlayerDAO> criteria = criteriaBuilder.createQuery(PlayerDAO.class);
final Root<PlayerDAO> playerRoot = criteria.from(PlayerDAO.class);
final Predicate siteCondition =
    criteriaBuilder.and(criteriaBuilder.equal(playerRoot.get(PlayerDAO_.site).get(SiteDAO_.name), "ausopen"),
    criteriaBuilder.equal(playerRoot.get(PlayerDAO_.site).get(SiteDAO_.year), 2016));

//create the search critera
criteria.select(playerRoot);
criteria.where(siteCondition);
final TypedQuery<PlayerDAO> query = manager.createQuery(criteria);
List<PlayerDAO> players = query.getResultList();
```

In Listing 4, we put together DAOs, JPA, and composite keys from the Java Persistence API, and we show an example query for all Australian 2016 players in our database. The steps here combine all of the pieces shown previously in this article. The JPA EntityManagerFactory parses the persistence.xml file and can create new connections to the data source, represented by the EntityManager object. An EntityManager represents a distinct connection to the database. It can be used to persist, update, or query to and from the database for data, which will be mapped to the Java objects, such as PlayerDAO. In Listing 4, we use the JPA criteria builder to build a SQL statement that will select all the rows from the PLAYERS table that are playing for the site that matches the name ausopen (Australian Open) and during the year 2016. This query will return multiple results, so query.getResultList returns a list of objects based on the metamodel from our enhanced JPA classes.

One challenge when using relational databases is that schema updates to tables can be a quite painful process, doubly so when it is necessary to retain the existing information in the table. Liquibase is a powerful tool that allows the creation of versioned and descriptive change sets in a markup format. We used the XML version of the Liquibase markup, although it also supports JSON and raw SQL change sets. Each change set is XML markup that describes the changes to apply to the database, whether it is as simple as inserting a new row into an existing table, or adding new indexes to a table, or crafting new columns for an existing table.
Listing 5. A change set that adds a new row to an existing table with liquibase

```xml
<changeSet author="boc" id="1.0.2">
  <comment>Add 2015 masters site</comment>
  <insert schemaName="eiblueus" tableName="sites">
    <column name="NAME" value="mast"/>
    <column name="YEAR" valueNumeric="2015"/>
    <column name="URL" value="http://www.masters.com/en_US/tournament/"/>
    <column name="LATITUDE" valueNumeric="33.4667"/>
    <column name="LONGITUDE" valueNumeric="81.9667"/>
    <column name="TOURNAMENT_SCHEDULE_START" valueDate="2015-04-09"/>
    <column name="TOURNAMENT_SCHEDULE_END" valueDate="2015-04-12"/>
    <column name="PRELIMINARY_SCHEDULE_START" valueDate="1970-01-01"/>
    <column name="PRELIMINARY_SCHEDULE_END" valueDate="1970-01-01"/>
  </insert>
</changeSet>
```

Listing 5 is a change set that shows Brian O'Connell adding the 2015 Masters tournament to the table named sites for the eiblueus schema of the database. Each of the columns has values for the change set, and is wrapped in the `<insert>` XML tag. Insertions are the most straightforward use of Liquibase, but inserting data using standard methods, such as JDBC or JPA connections, is straightforward as well.

Listing 6. Updating a table schema using a Liquibase change set

```xml
<changeSet author="cnmcavoy" id="1.0.3">
  <comment>Add twitter follower, klout, retweet data</comment>
  <addColumn schemaName="eiblueus" tableName="TWEET">
    <column name="KLOUT" type="INTEGER"/>
    <column name="FOLLOWERS" type="INTEGER"/>
    <column name="FRIENDS" type="INTEGER"/>
    <column name="RETWEETS" type="INTEGER"/>
  </addColumn>
  <createIndex indexName="I_TWEET_USER" schemaName="eiblueus" tableName="TWEET" unique="false">
    <column name="USER" type="varchar(255)"/>
  </createIndex>
</changeSet>
```

Listing 6 shows a more complex Liquibase change set that adds several new columns to a table, and then creates an index on that same table. These actions automatically have rollback commands associated with them, but if more customized behavior had been needed, it could have been specified in a `<rollback>` element inside of the change set. The default behavior would undo the creation of the new columns and index, dropping them from the table if an error occurred while modifying the schema. Each change set is applied as a single transaction, and the rollback occurs if the transaction fails. This allows developers to create discrete, small-schema updates that are each individually less dangerous than mass-table updates, and come with safety checks and more advanced rollback handling if desired.

Listing 7. A complete database changelog for Liquibase, including the previous examples

```xml
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog/1.9" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <changeSet author="boc" id="1.0.2">
    <comment>Add 2015 masters site</comment>
  </changeSet>
</databaseChangeLog>
```
<table>
<thead>
<tr>
<th>NAME</th>
<th>YEAR</th>
<th>URL</th>
<th>LATITUDE</th>
<th>LONGITUDE</th>
<th>TOURNAMENT_SCHEDULE_START</th>
<th>TOURNAMENT_SCHEDULE_END</th>
<th>PRELIMINARY_SCHEDULE_START</th>
<th>PRELIMINARY_SCHEDULE_END</th>
</tr>
</thead>
</table>

```xml
<changeSet author="cnmcavoy" id="1.0.3">
  <comment>Add twitter follower, klout, retweet data</comment>
  <addColumn schemaName="eiblueus" tableName="TWEET">
    <column name="FOLLOWERS" type="INTEGER"/>
    <column name="FRIENDS" type="INTEGER"/>
    <column name="RETWEETS" type="INTEGER"/>
  </addColumn>
</changeSet>

<changeSet author="cnmcavoy" id="1.0.4">
  <comment>Add twitter gnip rules</comment>
  <createTable schemaName="eiblueus" tableName="TweetDAO_gnipRules">
    <column name="TWEETDAO_ID" type="VARCHAR(255)"/>
    <column name="element" type="VARCHAR(255)"/>
  </createTable>
  <createIndex indexName="I_TWTDRLS_TWEETDAO_ID" schemaName="eiblueus" tableName="TweetDAO_gnipRules" unique="false">
    <column name="TWEETDAO_ID" type="varchar(255)"/>
  </createIndex>
</changeSet>

<changeSet author="cnmcavoy" id="1.0.5">
  <comment>Add player id index</comment>
  <createIndex indexName="I_PLAYERS_ID" schemaName="eiblueus" tableName="PLAYERS" unique="false">
    <column name="ID" type="varchar(255)"/>
  </createIndex>
</changeSet>

<changeSet author="boc" id="1.0.6">
  <comment>Add Reach Table and Indexes</comment>
  <createTable schemaName="eiblueus" tableName="REACH">
    <column name="INSTANT" type="BIGINT">
      <constraints nullable="false"/>
    </column>
    <column name="TYPE" type="VARCHAR(255)"/>
    <column name="REACH" type="BIGINT">
      <constraints nullable="false"/>
    </column>
  </createTable>
  <addPrimaryKey columnNames="INSTANT, TYPE" schemaName="eiblueus" tableName="REACH"/>
  <createIndex indexName="I_REACH_INSTANT" schemaName="eiblueus" tableName="REACH" unique="false">
    <column name="INSTANT" type="BIGINT"/>
  </createIndex>
</changeSet>
```

```xml
</databaseChangeLog>
```
IBM DB2 HADR and high availability

The DB2 aspect of the PCC system must be highly available in order to follow disaster-avoidance principles. Figure 1 shows the architecture of the database system. A two-sited geo-region architecture supported the PCC system. As a result, peer-to-peer replication between DB2s ensured that the content between geo-region P1 and geo-region P3 were in sync to provide data consistency. Each geo-region was replicated to a development server, z10095.

Each of the replicated databases were used for disaster avoidance. If one of the databases or both lost disk, the replicated database would be used to recreate the data state. Unidirectional replication from both P1 and P3 created a read-only database for streaming data. The development team could use real-time data from production servers to test big data algorithms and the visualization of real-time data in motion. As a result, data corruption or unsupported formats were detected before the deployment into production.

**Figure 1. The DB2 system was highly available during production use and mirrored for development.**

The PCC system required noSQL data storage where different types of relations within and between data elements were required. The data for PCC was generally trended over time, which made the case for a time series database. All of the application analytics that monitored the inner function of the application was also captured and aggregated during specified time steps. In the next section, we discuss the use of Graphite, StatsD, and a visualization tool called Grafana.
Graphite, StatsD, and Grafana

Graphite is a numeric time-series database and frontend that allows for simple instrumentation and measurements. Graphite has three main components:

- **Carbon.** A daemon that listens and accepts new data.
- **Whisper.** A database designed for storing time-series data efficiently.
- **The webapp.** Provides a RESTful API and rendering capabilities for graphing.

Information can be sent to the Carbon daemon through AMQP or simple UDP messages. The Whisper database for Graphite is a fixed-size database that compresses older metrics in favor of more recent higher-resolution metrics. The exact compression ratios of metrics can be configured in the storage schema. The default storage schema is shown in Listing 8.

**Listing 8. The default Carbon storage schema configuration settings**

```
[carbon]
pattern = ^carbon\.\.
retentions = 60:90d

[default_1min_for_1day]
pattern = .*
retentions = 60s:1d
```

The default configuration stores up to 60 data points that match the regular expression `^carbon\.\.` which Carbon uses for internal profiling of itself. All other data points match the `.*` expression, and each metric will be stored once every 60 seconds and retained for a single day. These defaults are fine for testing, but for larger analysis, longer retention periods are needed.

**Listing 9. The Predictive Cloud Computing system storage schema configuration settings**

```
[carbon]
pattern = ^carbon\.\.
retentions = 60:90d

[http]
pattern = http\.\.
retentions = 10s:1d,1m:30d,5m:90d

[tasks]
pattern = tasks\.\.
retentions = 60s:1d,5m:30d,10m:180d

[default]
pattern =.*
retentions = 60s:1d,5m:30d,10m:1y
```

The retentions depicted in Listing 9 allow longer retention, but also aggregate the data after time has passed to prevent the Whisper database from growing excessively large. For example, the HTTP stats are aggregated into 10s intervals for the first day, then merged into one-minute intervals for the next thirty days and five-minute intervals for the next ninety days. They are eventually dropped entirely from the database.
One of the other important aspects of Graphite is Statsd, a daemon that listens for information and creates meaningful statistical measurements such as counters, times, and gauges from simple UDP messages. For example, echo "test.metric:1:c | nc -u -w0 127.0.0.1 8125" will send a UDP message to the Statsd daemon on the local machine, incrementing the counter for the metric named 'test.metric'. The java-statsd-client library makes it similarly simple to send metrics to Statsd and Graphite from the PCC system.

Listing 10. How to use the java-statsd-client to send data to StatsD and Graphite

```java
StatsDClient stats = new NonBlockingStatsDClient("test", "127.0.0.1", 8125);
stats.incrementCounter("metric");
stats.close()
```

Inside the PCC system, all HTTP requests were instrumented to measure the time to fulfill the request and whether the request returned a 2XX (successful) or error (4XX or 5XX) HTTP status code in response to the request. The JavaX servlet filter made it very simple to intercept the requests to each of our JaxRS endpoints in our application.

Listing 11. The JavaX Filter that instrumented HTTP calls for the Predictive Cloud Computing system

```java
public class GraphiteStatusFilter implements Filter {
    private TournamentConfiguration config;

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        this.config = (TournamentConfiguration)
            filterConfig.getServletContext().getAttribute(AbstractBigEngineJob.CONFIG_CONTEXT_KEY);
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse resp, FilterChain chain) throws IOException,
    ServletException {
        final StatsDClient stats;
        if (request instanceof HttpServletRequest) {
            HttpServletRequest req = (HttpServletRequest)request;
            stats = getStatsDClient(req);
        } else {
            stats = new NoOpStatsDClient();
        }

        stats.increment("request");
        final Stopwatch time = Stopwatch.createStarted();
        try {
            chain.doFilter(request, resp);
            if (resp instanceof HttpServletResponse) {
                HttpServletResponse response = (HttpServletResponse)resp;
                if (response.getStatus() / 100 == 2) {
                    stats.increment("success");
                } else {
                    stats.increment("failure");
                }
            }
        } catch (IOException e) {
            stats.increment("failure");
            throw e;
        } catch (ServletException e) {
            stats.increment("failure");
            throw e;
        }
    }
}
```
Once statistics are being collected by the instrumented codebase, the values are analyzed or visualized for human consumption. Graphite's webapp provides a limited graph-rendering front end, but there are more powerful open-source front ends for the Graphite data, such as Grafana. Grafana is a dashboard-creation tool for metrics, which can plug into a number of back-end databases, including Graphite.

**Figure 2. A Grafana dashboard that shows server information**
Graphite for feature extraction visualization

The predictive modeling within the PCC system applied a multiple regression model that was trained over a diverse set of independent variables. The input for the model were features that were extracted from a simulated game tournament into the future that included game play statistics, social popularity, and web server behavior. The output of the model produced a single numerical representation of an expected origin server load. Overall, the model was combined with traditional cyclical forecasting to account for sharp spikes or periods of high demand for tennis or golf content.

Each of the feature extractors sent count and timing statistics to Graphite. For example, each time the Tennis or Golf Twitter Count Factor was applied to a simulated game tournament, a time and count metric was pushed into the Graphite data store. Each of the algorithms was wrapped into the Unstructured Information Management Architecture—Asynchronous Scaleout open-source project. As a result, many of the algorithms ran in parallel, which caused a large number of messages to be sent to Graphite.

Listing 12. An algorithm to forecast the number of tweets per player for a player within a tournament at a future time

```java
public void process(JCas jCas) throws AnalysisEngineProcessException {
    logger.debug("UIMA: Running Twitter Count Factor");
    StatsDClient statsCount = new NoOpStatsDClient();
    StatsDClient statsRunTime = new NoOpStatsDClient();
    StatsDClient statsContributeCount = new NoOpStatsDClient();
    TournamentConfiguration tournamentConfig = null;
    AnnotationIndex<Annotation> dataAggIndex =
        jCas.getAnnotationIndex(com.ibm.ei.zepplin.factors.common.model.RemoteDataAggregator.type);
    Iterator<Annotation> dataAggIt = dataAggIndex.iterator();
    if (dataAggIt.hasNext()) {
        RemoteDataAggregator remoteProps = (RemoteDataAggregator) dataAggIt.next();
        String countTag = FactorState.buildTag(remoteProps.getHostTag(), FactorState.SOCIAL,
            FactorState.COUNT);
        statsCount = getStatsDClient(countTag, remoteProps.getHostName(), remoteProps.getHostPort());
        String runTimeTag = FactorState.buildTag(remoteProps.getHostTag(), FactorState.SOCIAL,
            FactorState.RUNTIME);
        statsRunTime = getStatsDClient(runTimeTag, remoteProps.getHostName(), remoteProps.getHostPort());
        String runContribCountTag = FactorState.buildTag(remoteProps.getHostTag(),
                remoteProps.getHostTag(), remoteProps.getHostTag(),
            FactorState.SOCIAL,
            FactorState.COUNT);
        statsContributeCount = getStatsDClient(runContribCountTag, remoteProps.getHostName(),
            remoteProps.getHostPort());
    }
    Stopwatch stopwatch = Stopwatch.createStarted();
    ...
    stopwatch.stop();
    statsRunTime.recordExecutionTime(this.getClass().getSimpleName(), (int)
        (stopwatch.elapsed(TimeUnit.MILLISECONDS)));
    statsCount.increment(this.getClass().getSimpleName());
    statsContributeCount.increment(this.getClass().getSimpleName());
}
```

Each golf event utilized a total of 15 different feature extractors while tennis implemented 18 diverse algorithms. The resulting data trended over time provided two functions. The first was a visual for end users and system administrators to understand the processing percentages and times of each algorithm. Figure 3 depicts the visualization where the bubble chart shows processing time and the doughnut plot is percentage of messages to each type of algorithm.
Figure 3. A doughnut plot showing the percentage of messages within UIMA-AS

The second function was to debug the application. Grafana, mentioned earlier in this article, provided runtime and traffic summaries for each algorithm. If an algorithm was running too long, the code could be receiving too many messages such that UIMA-AS needed to be redistributed across machines. Alternatively, an algorithm was optimized on several occasions to ensure near real-time operations. Other types of system and code level errors were discovered if traffic was zero or relatively decreasing when contrasted to other algorithms. The symptom generally meant an algorithm was paging and needed more memory or that a memory leak was impairing the feature extractor.

Conclusion

In this tutorial, we have shown how the PCC used a SQL-based store such as DB2 and a noSQL store such as Graphite. Within code, the use of JPA2 and Liquibase enabled database changes to be traceable, repeatable, and reliable. The noSQL data store, Graphite, was invaluable as a time series database that supported analytic components of PCC and for diagnosing problematic runtime symptoms. Several examples, code blocks, and configurations depict tangible implementation details.

In part 8 of the series, we will discuss the distributed feature extraction and predictive model used throughout the PCC system.
Related topics

- Liquibase [video]
- IBM DB2
- Java Persistence API
- Graphite
- IBM Predicts Cloud Computing Demand for Sports Tournaments

© Copyright IBM Corporation 2016
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)