Google App Engine and Datastore

Presentation by Ivan Kuzmenko
What Is Google App Engine?

- Lets you run web applications on Google's infrastructure.

- App Engine applications are easy to build, easy to maintain, and easy to scale as your traffic and data storage needs grow. You just upload your application, and it's ready to serve your users.

- Supports apps written in several programming languages: Java, Java servlets, JavaScript, Ruby, Python and Go.

- App Engine costs nothing to get started. All applications can use up to 1 GB of storage and enough CPU and bandwidth to support an efficient app serving around 5 million page views a month, absolutely free.
The Application Environment

- Dynamic web serving, with full support for common web technologies.
- Persistent storage with queries, sorting and transactions.
- Automatic scaling and load balancing.
- APIs for authenticating users and sending email using Google Accounts.
- A fully featured local development environment that simulates Google App Engine on your computer.
- Task queues for performing work outside of the scope of a web request.
Datastore Overview
Introducing the Datastore

• The App Engine datastore saves data objects, known as entities. An entity has one or more properties, named values of one of several supported data types.

• The datastore can execute multiple operations in a single transaction. This is especially useful for distributed web applications, where multiple users may be accessing or manipulating the same data at the same time.

• Unlike traditional databases, the datastore uses a distributed architecture to automatically manage scaling to very large data sets. It is very different from a traditional relational database in how it describes relationships between data objects.

• Two entities of the same kind can have different properties. Different entities can have properties with the same name, but different value types. While the datastore interface has many of the same features of traditional databases, the datastore’s unique characteristics imply a different way of designing and managing data to take advantage of the ability to scale automatically.
App Engine provides two different data storage options

• The Master/Slave datastore uses a master-slave replication system, which asynchronously replicates data as you write it to a physical data center. Since only one data center is the master for writing at any given time, this option offers strong consistency for all reads and queries, at the cost of periods of temporary unavailability during data center issues or planned downtime.

• The High Replication datastore (HRD) is the default for new applications. The HRD is a highly available, highly reliable storage solution. It remains available for reads and writes during planned downtime and is extremely resilient in the face of catastrophic failure - but it costs more than the master/slave option. In the High Replication datastore, data is replicated across data centers using a system based on the Paxos algorithm.
### Comparing the Data Storage Options

<table>
<thead>
<tr>
<th></th>
<th>High Replication</th>
<th>Master/Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage</td>
<td>1x</td>
<td>1/3x</td>
</tr>
<tr>
<td>Put/Delete CPU</td>
<td>1x</td>
<td>5/8x</td>
</tr>
<tr>
<td>Get CPU</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>Query CPU</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put/Delete Latency</td>
<td>1/2x–1x</td>
<td>1x</td>
</tr>
<tr>
<td>Get Latency</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>Query Latency</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get/Put/Delete</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Most Queries</td>
<td>Eventual</td>
<td>Strong</td>
</tr>
<tr>
<td>Ancestor Queries</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Occasional Planned Read-Only Period</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Unplanned Downtime</td>
<td>Extremely rare. No data loss.</td>
<td>Rare. Possible to lose a small % of writes that occurred near the downtime (recoverable after event).</td>
</tr>
<tr>
<td>Python 2.7 support</td>
<td>The only supported datastore in Python 2.7</td>
<td>Not supported in Python 2.7</td>
</tr>
</tbody>
</table>
Differences From SQL

• The App Engine datastore is designed to scale.

• No joins are supported in the datastore.

• The datastore also does not allow inequality filtering on multiple properties or filtering of data based on results of a sub-query.

• It doesn't require data kinds to have a consistent property set.

• And it can either return entire entities or only entity keys from a query.
Entities And Keys

- The App Engine datastore is best understood as an object database.
- Each data record is an entity, represented in code as an object.
- Each entity has a key that uniquely identifies the entity across all entities in the datastore.
- Each entity has one or more named properties, represented as attributes of the object.
A key has several components:

- The kind of the entity.
- An identifier that is either a name assigned to the entity by the app or a numeric ID assigned by the datastore.
- A path, an optional part that specifies another entity as this entity's parent.

```
Entity employee = new Entity("Employee", "asalieri");
```

```
Entity employee = new Entity("Employee");
```
This example prepares a new datastore entity of the kind "Employee", and sets several properties with different types of values. It then tells the datastore to store the new entity.

```java
import java.util.Date;
import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;

// ...
DatastoreService datastore = DatastoreServiceFactory.get(DatastoreServiceFactory().getService());

Entity employee = new Entity("Employee");
employee.setProperty("firstName", "Antonio");
employee.setProperty("lastName", "Salieri");
Date hireDate = new Date();
employee.setProperty("hireDate", hireDate);
employee.setProperty("attendedHrTraining", true);

datastore.put(employee);
```
Entity Groups And Ancestor Path

- To create an entity in a group, you declare that another entity is the parent of the new entity when you create it.

- An entity created without a parent is a root entity. A root entity without any children is an entity group by itself.

- Each entity's key contains a path of entities starting with the entity group root.

- This path is an essential part of the entity's complete key. A complete key can be represented by the kind and ID (or key name) of each entity in the path.

- A chain of parent entities from an entity up to the root is the path for the entity, and members of the path are the entity's ancestors.

- The parent of an entity is defined when the entity is created, and cannot be changed later.
To specify that an entity be created in an existing entity group, provide the Key of the parent entity as an argument to the Entity constructor of the new entity. You can get a Key object by calling the getKey() method on the parent Entity.

```java
Entity employee = new Entity("Employee");
datastore.put(employee);
Entity address = new Entity("Address", employee.getKey());
```

If the entity with the parent also has a key name, provide the key name (a String) as the second argument, and the Key of the parent entity as the third argument.

```java
Entity address = new Entity("Address", "addr1", employee.getKeY());
```

The complete key of an entity is the key of the parent, if any, followed by the kind of the entity, followed by the key name or system ID.

Employee:8261 / Address:1
Properties

The data for each entity is stored in one or more properties. Each property has a name and at least one value. Each value is of one of several supported data types, such as Unicode string, integer, date-time, or byte string.

<table>
<thead>
<tr>
<th>Value type</th>
<th>Java type</th>
<th>Sort order</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>boolean or java.lang.Boolean</td>
<td>false &lt; true</td>
<td></td>
</tr>
<tr>
<td>Byte string, short</td>
<td>com.google.appengine.api.datastore.ShortBlob</td>
<td>byte order</td>
<td>Up to 500 bytes. A value longer than 500 bytes throws a JDOFatalUserException.</td>
</tr>
<tr>
<td>Byte string, long</td>
<td>com.google.appengine.api.datastore.Blob</td>
<td>n/a</td>
<td>up to 1 megabyte; not indexed</td>
</tr>
<tr>
<td>category</td>
<td>com.google.appengine.api.datastore.Category</td>
<td>Unicode</td>
<td></td>
</tr>
<tr>
<td>Date and time</td>
<td>java.util.Date</td>
<td>chronological</td>
<td></td>
</tr>
<tr>
<td>email address</td>
<td>com.google.appengine.api.datastore.Email</td>
<td>Unicode</td>
<td></td>
</tr>
<tr>
<td>floating point number</td>
<td>float, java.lang.Float, double, java.lang.Double</td>
<td>numeric</td>
<td>64-bit double precision, IEEE 754</td>
</tr>
<tr>
<td>geographical point</td>
<td>com.google.appengine.api.datastore.GeoPt</td>
<td>by latitude, then longitude</td>
<td></td>
</tr>
<tr>
<td>Google Accounts user</td>
<td>com.google.appengine.api.users.User</td>
<td>email address in Unicode order</td>
<td></td>
</tr>
</tbody>
</table>
Saving, Getting, and Deleting Entities

With the Java datastore API, you save, get, and delete entities using methods of the `DatastoreService`. You get this object by calling the `getDatastoreService()` class method of the `DatastoreServiceFactory` class.

```java
import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;

// ...
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
```

To create or update an entity in the datastore, call the `put()` method with the `Entity` object to save.

```java
Entity employee = new Entity("Employee");
// ... set properties ...

datastore.put(employee);
```
Saving, Getting, and Deleting Entities

To get an entity identified by a given key, call the `get()` method with a `Key` object.

```java
// Key employeeKey = ...;
Entity employee = datastore.get(employeeKey);
```

To delete an entity with a given key, call the `delete()` method with a `Key` object.

```java
// Key employeeKey = ...;
datastore.delete(employeeKey);
```

Note: The datastore API does not distinguish between creating a new entity and updating an existing entity. If the object's key represents an entity that exists, calling the `put()` method with the object overwrites the entity. You can use a `transaction` to test whether an entity with a given key exists before creating one.
Making Keys

The `KeyFactory` class can create `Key` objects from known components, such as the kind and the key name.

```java
Key k = KeyFactory.createKey(Employee.class.getSimpleName(), "Alfred.Smith@example.com");
```

To recreate a key of an entity of kind "Employee" with a system-assigned numeric ID of 52234 (and no entity group parent):

```java
Key k = KeyFactory.createKey(Employee.class.getSimpleName(), 52234);
```

If the entity associated with a key has an entity group parent, you can create the key with the `KeyFactory.Builder` class:

```java
Key k = new KeyFactory.Builder(Employee.class.getSimpleName(), 52234)
    .addChild(ExpenseReport.class.getSimpleName(), "A23279").getKey();
```

The Builder instance's `addChild()` method returns the Builder, so you can chain calls to add each element of the key path. To get the complete Key value for a given builder, call the Builder's `getKey()` method.
Making Keys

The `KeyFactory` class also includes the class methods `keyToString()` and `stringToKey()` for converting keys to and from a string representation. (Note that this is different from the Key class's `toString()` method, which returns a human-readable value suitable for debugging.) The string version of a key value is "web safe": it does not contain characters considered special in HTML or in URLs.

```java
String employeeKeyStr = KeyFactory.keyToString(employeeKey);
// ...

Key employeeKey = KeyFactory.stringToKey(employeeKeyStr);
Entity employee = datastore.get(employeeKey);
```

Be aware that `Key.toString()` does not return a machine-parsable String representation of the Key, but rather a human-readable string useful for debugging and logging. If you need a String that can be converted to a Key, use `KeyFactory.keyToString(key)`. 
Batch Operations

The `put()`, `get()`, and `delete()` methods accept `java.lang.Iterable Entity` objects (for `put()`) and `Key` objects (for `get()` and `delete()`). This performs the action on multiple entities in a single datastore call. The batch operation groups all the entities/keys by entity group, then performs operations on each entity group in parallel.

A batch call to the datastore is faster than making separate calls for each entity because it only incurs the overhead of one service call, and, if there are multiple entity groups involved, the work for each entity group is performed on the server side in parallel.

```java
import java.util.Arrays;
import java.util.List;

// ...
Entity employee1 = new Entity("Employee");
Entity employee2 = new Entity("Employee");
Entity employee3 = new Entity("Employee");
// ...
List<Entity> employees = Arrays.asList(employee1, employee2, employee3);
datastore.put(employees);
```

A batch call to `put()` or `delete()` may succeed for some entities but not others. If it is important that the call succeed completely or fail completely, you must use a `transaction`, and all affected entities must be in the same entity group. Attempting a batch operation inside a transaction with entities or keys that belong to multiple entity groups results in an `IllegalArgumentException`. 
Queries

A query retrieves datastore entities that meet a specified set of conditions. The query specifies an entity kind, zero or more conditions based on entity property values (sometimes called "filters"), and zero or more sort order descriptions. When the query is executed, it fetches all entities of the given kind that meet all of the given conditions, sorted in the order described.

The Master/Slave Datastore and the High Replication Datastore have different guarantees when it comes to query consistency. By default:

- The Master/Slave datastore is strongly consistent for all queries.
- The High Replication datastore is strongly consistent by default for queries within an entity group. With the High Replication Datastore, non-ancestor queries are always eventually consistent.
Queries

The low-level Java API provides a `Query` class for constructing queries and a `PreparedQuery` class for fetching and returning entities from the datastore.

```java
import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;
import com.google.appengine.api.datastore.PreparedQuery;
import com.google.appengine.api.datastore.Query;

// ...
// Get the Datastore Service
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();

// The Query interface assembles a query
Query q = new Query("Person");
q.addFilter("lastName", Query.FilterOperator.EQUAL, lastNameParam);
q.addFilter("height", Query.FilterOperator.LESS_THAN, maxHeightParam);

// PreparedQuery contains the methods for fetching query results
// from the datastore
PreparedQuery pq = datastore.prepare(q);

for (Entity result : pq.asIterable()) {
    String firstName = (String) result.getProperty("firstName");
    String lastName = (String) result.getProperty("lastName");
    Long height = (Long) result.getProperty("height");
    System.out.println(lastName + " " + firstName + ", " + height.toString() + " inches tall");
}
```
Queries

The filter operator can be any of the following:
Query.FilterOperator.LESS_THAN
Query.FilterOperator.LESS_THAN_OR_EQUAL
Query.FilterOperator.EQUAL
Query.FilterOperator.GREATER_THAN
Query.FilterOperator.GREATER_THAN_OR_EQUAL
Query.FilterOperator.NOT_EQUAL
Query.FilterOperator.IN (equal to any of the values in the provided list)

• The NOT_EQUAL operator actually performs two queries: one where all other filters are the same and the not-equal filter is replaced with a less-than filter, and one where the not-equal filter is replaced with a greater-than filter.
• The IN operator also performs multiple queries, one for each item in the provided list value where all other filters are the same and the IN filter is replaced with an equal-to filter.
• A single query containing NOT_EQUAL or IN operators is limited to 30 sub-queries.

A query can also return just the keys of the result entities instead of the entities themselves.

```java
Query q = new Query("Person").setKeyOnly();
```
Ancestor Queries

You can filter your datastore queries to a specified ancestor, so that the result set only includes entities descended from the specified ancestor. In other words, all of the results will have the ancestor as their parent, or parent's parent, or etc.

```java
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Entity person = new Entity("Person", "tom");

Entity weddingPhoto = new Entity("Photo", person.getKey());
    weddingPhoto.setProperty("imageUrl",
        "http://domain.com/some/path/to/wedding_photo.jpg");

Entity babyPhoto = new Entity("Photo", person.getKey());
    babyPhoto.setProperty("imageUrl",
        "http://domain.com/some/path/to/baby_photo.jpg");

Entity dancePhoto = new Entity("Photo", person.getKey());
    dancePhoto.setProperty("imageUrl",
        "http://domain.com/some/path/to/dance_photo.jpg");

Entity campingPhoto = new Entity("Photo");
    campingPhoto.setProperty("imageUrl",
        "http://domain.com/some/path/to/camping_photo.jpg");

datastore.put(Arrays.asList(person, weddingPhoto, babyPhoto, dancePhoto, campingPhoto));

Query userPhotosQuery = new Query("Photo");
userPhotosQuery.setAncestor(person.getKey());

// This returns weddingPhoto, babyPhoto and dancePhoto, but not campingPhoto because tom is not an ancestor.
List<Entity> results = datastore.prepare(userPhotosQuery).asList(
    FetchOptions.Builder.withDefaults());
```
Restrictions on Queries

- Filtering or sorting on a property requires that the property exists.
- Inequality filters are allowed on one property only.

```java
import com.google.appengine.api.datastore.Query;
import com.google.appengine.api.datastore.Query.FilterOperator;
import com.google.appengine.api.datastore.Query.SortDirection;

Query q = new Query("Person");
q.addFilter("birthYear", FilterOperator.GREATER_THAN_OR_EQUAL, minBirthYearParam);
q.addFilter("birthYear", FilterOperator.LESS_THAN_OR_EQUAL, maxBirthYearParam);

Query q = new Query("Person");
q.addFilter("birthYear", FilterOperator.GREATER_THAN_OR_EQUAL, minBirthYearParam);
// Error
q.addFilter("height", FilterOperator.GREATER_THAN_OR_EQUAL, minHeightParam);

Query q = new Query("Person");
q.addFilter("lastName", FilterOperator.EQUAL, lastNameParam);
q.addFilter("city", FilterOperator.EQUAL, cityParam);
q.addFilter("birthYear", FilterOperator.GREATER_THAN_OR_EQUAL, minBirthYearParam);
```
Restrictions on Queries

• Properties in inequality filters must be sorted before other sort orders

```java
Query q = new Query("Person");
q.addFilter("birthYear", FilterOperator.GREATER_THAN_OR_EQUAL, minBirthYearParam);
q.addSort("lastName", SortDirection.ASCENDING); // ERROR
```

```java
Query q = new Query("Person");
q.addFilter("birthYear", FilterOperator.GREATER_THAN_OR_EQUAL, minBirthYearParam);
q.addSort("lastName", SortDirection.ASCENDING); // ERROR
q.addSort("birthYear", SortDirection.ASCENDING);
```

```java
Query q = new Query("Person");
q.addFilter("birthYear", FilterOperator.GREATER_THAN_OR_EQUAL, minBirthYearParam);
q.addSort("birthYear", SortDirection.ASCENDING);
q.addSort("lastName", SortDirection.ASCENDING);
```
Fetching Results

• If you want to return only a single entity matching your query, you can use the `PreparedQuery.asSingleEntity()` method. This will return the first result that matches the query.

• You can specify a limit and offset with your query to control the number and range of results returned in one batch. Specifying an integer limit returns up to that number of results that match the query. Using an integer offset will skip that number of results and return the rest, up to the limit specified.

```java
// Construct then prepare your query
List<Entity> get5TallestPeople() {
    DatastoreService ds = DatastoreServiceFactory.getDatastoreService();
    Query q = new Query("Person");
    q.addSort("height", SortDirection.DESCENDING);
    PreparedQuery pq = ds.prepare(q);
    return pq.asList(P FetchOptions.Builder.withLimit(5));
}
```

```java
return pq.asList(FetchOptions.Builder.withLimit(5).offset(5));
```
Using Transactions

- A *transaction* is a set of datastore operations on one or more entity. Each transaction is guaranteed to be atomic, which means that transactions are never partially applied. Either all of the operations in the transaction are applied, or none of them are applied.

- An operation may fail due to a high rate of contention when too many users try to modify an entity group at the same time. Or an operation may fail due to the application reaching a quota limit. Or there may be an internal error with the datastore. In all cases, the datastore API raises an exception.

- Transactions are an optional feature of the datastore; you're not required to use transactions to perform datastore operations.

```java
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Transaction txn = datastore.beginTransaction();
try {
    Key employeeKey = KeyFactory.createKey("Employee", "Joe");
    Entity employee = datastore.get(employeeKey);
    employee.setProperty("vacationDays", 10);
    datastore.put(employee);
    txn.commit();
} finally {
    if (txn.isActive()) {
        txn.rollback();
    }
}
```
Entity Groups

• Every entity belongs to an entity group, a set of one or more entities that can be manipulated in a single transaction. Entity group relationships tell App Engine to store several entities in the same part of the distributed network. A transaction sets up datastore operations for an entity group, and all of the operations are applied as a group, or not at all if the transaction fails.

• When the application creates an entity, it can assign another entity as the parent of the new entity. Assigning a parent to a new entity puts the new entity in the same entity group as the parent entity.

• An entity without a parent is a root entity. An entity that is a parent for another entity can also have a parent. A chain of parent entities from an entity up to the root is the path for the entity, and members of the path are the entity's ancestors. The parent of an entity is defined when the entity is created, and cannot be changed later.

• Every entity with a given root entity as an ancestor is in the same entity group. All entities in a group are stored in the same datastore node. A single transaction can modify multiple entities in a single group, or add new entities to the group by making the new entity's parent an existing entity in the group.
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Entity person = new Entity("Person", "tom");
datastore.put(person);

// Transactions on root entities
Transaction tx = datastore.beginTransaction();

Entity tom = datastore.get(person.getKey());
tom.setProperty("age", 40);
datastore.put(tom);
tx.commit();

// Transactions on child entities
tx = datastore.beginTransaction();
tom = datastore.get(person.getKey());
Entity photo = new Entity("Photo", tom.getKey());

// Create a Photo that is a child of the Person entity named "tom"
photo.setProperty("photoUrl", "http://domain.com/path/to/photo.jpg");
datastore.put(photo);
tx.commit();

// Transactions on entities in different entity groups
tx = datastore.beginTransaction();
tom = datastore.get(person.getKey());
Entity photoNotAChild = new Entity("Photo");
photoNotAChild.setProperty("photoUrl", "http://domain.com/path/to/photo.jpg");
datastore.put(photoNotAChild);

// Throws IllegalArgumentException because the Person entity
// and the Photo entity belong to different entity groups.
tx.commit();
Java Data Objects (JDO)
Using JDO with App Engine

• Java Data Objects (JDO) is a standard interface for storing objects containing data into a database.

• The standard defines interfaces for annotating Java objects, retrieving objects with queries, and interacting with a database using transactions.

• An application that uses the JDO interface can work with different kinds of databases without using any database-specific code, including relational databases, hierarchical databases, and object databases.

• As with other interface standards, JDO simplifies porting your application between different storage solutions.

• The App Engine Java SDK includes an implementation of JDO 2.3 for the App Engine datastore. The implementation is based on DataNucleus Access Platform, the open source reference implementation for JDO 2.3.
Getting a PersistenceManager Instance

```java
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;

public final class PMF {
    private static final PersistenceManagerFactory pmfInstance =
        JDOHelper.getPersistenceManagerFactory("transactions-optimal");

    private PMF() {}

    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }
}
```

```java
import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;

import PMF;

// ...
PersistenceManager pm = PMF.get().getPersistenceManager();
```

```java
try {
    // ... do stuff with pm ...
} finally {
    pm.close();
}
```
Defining Data Classes with JDO

Each object saved by JDO becomes an entity in the App Engine datastore. The entity's kind is derived from the simple name of the class. Each persistent field of the class represents a property of the entity, with the name of the property equal to the name of the field.

```java
import javax.jdo.annotations.PersistenceCapable;

@PersistenceCapable
public class Employee {
    // ...
}

import java.util.Date;
import javax.jdo.annotations.Persistent;

// ...
    @Persistent
    private Date hireDate;
```

A data class must have one and only one field dedicated to storing the primary key of the corresponding datastore entity.

```java
import com.google.appengine.api.datastore.Key;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PrimaryKey;

// ...
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;
```
Defining Data Classes with JDO

Embedded classes allow you to model a field value using a class without creating a new datastore entity and forming a relationship. The fields of the object value are stored directly in the datastore entity for the containing object.

```java
import javax.jdo.annotations.Embedded;
import javax.jdo.annotations.EmbeddedOnly;

// ... imports ...

@PersistenceCapable
public class EmployeeContacts {
  @PrimaryKey
  @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
  Key key;
  @PersistenceCapable
  @EmbeddedOnly
  public static class ContactInfo {
    @Persistent
    private String streetAddress;

    @Persistent
    private String city;

    @Persistent
    private String stateOrProvince;

    @Persistent
    private String zipCode;

    // ... accessors ...
  }

  @Persistent
  @Embedded
  private ContactInfo homeContactInfo;
}
```
Defining Data Classes with JDO

Inheritance

```java
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.Inheritance;
import javax.jdo.annotations.InheritanceStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable
@Inheritance(strategy = InheritanceStrategy.SUBCLASS_TABLE)
public abstract class Worker {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private String department;
}
```

```java
import java.util.Date;
// ... imports ...

@PersistenceCapable
public class Employee extends Worker {
    @Persistent
    private int salary;
}

@PersistenceCapable
@Inheritance(customStrategy = "complete-table")
public class FormerEmployee extends Employee {
    @Persistent
    private Date lastDay;
}
```
Creating, Getting and Deleting Data in JDO

Making Objects Persistent

```java
PersistenceManager pm = PMF.get().getPersistenceManager();

Employee e = new Employee("Alfred", "Smith", new Date());

try {
    pm.makePersistent(e);
} finally {
    pm.close();
}
```

Getting an Object by Key

```java
Key k = KeyFactory.createKey(Employee.class.getSimpleName(), "Alfred-Smith@example.com");
Employee e = pm.getObjectById(Employee.class, k);  
Employee e = pm.getObjectById(Employee.class, "Alfred-Smith@example.com");
```
Creating, Getting and Deleting Data in JDO

Updating an Object

```java
public void updateEmployeeTitle(User user, String newTitle) {
    PersistenceManager pm = PMF.get().getPersistenceManager();
    try {
        Employee e = pm.getObjectById(Employee.class, user.getEmail());
        if (titleChangeIsAuthorized(e, newTitle)) {
            e.setTitle(newTitle);
        } else {
            throw new UnauthorizedTitleChangeException(e, newTitle);
        }
    } finally {
        pm.close();
    }
}
```

Deleting an Object

```java
pm.deletePersistent(e);
```
Entity Relationships in JDO

Owned One-to-One Relationships

```java
import com.google.appengine.api.datastore.Key;
// ... imports ...

@PersistenceCapable
class ContactInfo {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private String streetAddress;
    // ...
}
```

```java
import ContactInfo;
// ... imports ...

@PersistenceCapable
class Employee {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private ContactInfo contactInfo;

    ContactInfo getContactInfo() {
        return contactInfo;
    }

    void setContactInfo(ContactInfo contactInfo) {
        this.contactInfo = contactInfo;
    }
    // ...
}
```

ContactInfo.java
Owned One-to-Many Relationships

**Employee.java**

```java
import java.util.List;

// ...
@Persistent
private List<ContactInfo> contactInfoSets;
```

A one-to-many bidirectional relationship is similar to a one-to-one, with a field on the parent class using the annotation `@Persistent(mappedBy = "...")`, where the value is the name of the field on the child class:

```java
import java.util.List;

// ...
@Persistent(mappedBy = "employee")
private List<ContactInfo> contactInfoSets;
```

**ContactInfo.java**

```java
import Employee;

// ...
@Persistent
private Employee employee;
```
Queries in JDO

JDO includes a query language for retrieving objects that meet a set of criteria. This language, called JDOQL, refers to JDO data classes and fields directly, and includes type checking for query parameters and results. JDOQL is similar to SQL, but is more appropriate for object-oriented databases like the App Engine datastore. (The App Engine datastore does not support SQL queries with the JDO interface.)

```java
import java.util.List;
import javax.jdo.Query;

// ...

Query query = pm.newQuery(Employee.class);
query.setFilter("lastName == lastNameParam");
query.setOrdering("hireDate desc");
query.declareParameters("String lastNameParam");

try {
    List<Employee> results = (List<Employee>) query.execute("Smith");
    if (!results.isEmpty()) {
        for (Employee e : results) {
            // ...
        }
    } else {
        // ... no results ...
    }
} finally {
    query.closeAll();
}
Queries in JDO

Here is the same query using the string syntax:

```java
Query query = pm.newQuery("select from Employee " +
   "where lastName == lastNameParam " +
   "parameters String lastNameParam " +
   "order by hireDate desc");

List<Employee> results = (List<Employee>) query.execute("Smith");
```

You can mix these styles of defining the query:

```java
Query query = pm.newQuery(Employee.class,
   "lastName == lastNameParam order by hireDate desc");
query.declareParameters("String lastNameParam");

List<Employee> results = (List<Employee>) query.execute("Smith");
```

The JDOQL string syntax supports value literals within the string for string values and numeric values. Surround strings in either single-quotes ('') or double-quotes (""). All other value types must use parameter substitution.

```java
Query query = pm.newQuery(Employee.class,
   "lastName == 'Smith' order by hireDate desc");
```
Queries in JDO

Query Filters:

A **filter** specifies a field name, an operator, and a value. The value must be provided by the app; it cannot refer to another field, or be calculated in terms of other fields.

The filter operator can be any of the following:
- < less than
- <= less than or equal to
- = equal to
- > greater than
- >= greater than or equal to
- != not equal to

contains() filters (more commonly known as IN filters in SQL) are also supported using the following syntax:

```java
// Give me all employees with lastName equal to Smith or Jones
Query query = pm.newQuery(Employee.class,
    "@p.contains(lastName)");
query.execute( Arrays.asList("Smith", "Jones") );
```
Queries in JDO

Query Filters:

Due to the way the App Engine datastore executes queries, a single query cannot use inequality filters (< <= >= > !=) on more than one property. Multiple inequality filters on the same property (such as querying for a range of values) are permitted.

```java
query.setFilter("lastName == 'Smith' && hireDate > hireDateMinimum");
query.declareParameters("Date hireDateMinimum");
```

In the JDOQL string syntax, you can separate multiple filters with || (logical "or") and && (logical "and"), although keep in mind that || can only be employed when the filters it separates all have the same field name. In other words, || is only legal in situations where the filters it separates can be combined into a single contains() filter:

```java
// legal, all filters separated by || are on the same field
Query query = pm.newQuery(Employee.class,
    "(lastName == 'Smith' || lastName == 'Jones') + "
    " && firstName == 'Harold'");

// not legal, filters separated by || are on different fields
Query query = pm.newQuery(Employee.class,
    "lastName == 'Smith' || firstName == 'Harold'");
```
Queries in JDO

Query Sort Orders

```java
query.setOrdering("hireDate desc, firstName asc");
```

Query Ranges

```java
query.setRange(5, 10);
```

Querying Entity Keys

```java
Food chocolate = ... 
Query q = pm.newQuery(Person.class);
qurey.setFilter("favoriteFood = favoriteFoodParam");
query.declareParameters(Key.class.getName() + " favoriteFoodParam");
List<Person> chocolateLovers = (List<Person>) q.executeUpdate(chocolate.getKey());
```

Deleting Entities By Query

```java
Query query = pm.newQuery(Person.class);
query.setFilter("height > maxHeightParam");
query.declareParameters("int maxHeightParam");
query.deletePersistentAll(maxHeight);
```
Unsupported Features of JDO

• Unowned relationships. You can implement unowned relationships using explicit Key values. JDO's syntax for unowned relationships may be supported in a future release.

• Owned many-to-many relationships.

• "Join" queries. You cannot use a field of a child entity in a filter when performing a query on the parent kind.

• JDOQL grouping and other aggregate queries.

• Polymorphic queries. You cannot perform a query of a class to get instances of a subclass. Each class is represented by a separate entity kind in the datastore.

• There is currently a bug preventing owned one-to-many relationships where the parent and the child are the same class, making it difficult to model tree structures. This will be fixed in a future release. You can work around this by storing explicit Key values for either the parent or children.