Building scalable, complex apps on App Engine

Brett Slatkin
May 27th, 2009
Agenda

- List properties
  - What they are, how they work
  - Example: Microblogging
  - Maximizing performance

- Merge-join
  - What it is, how it works; list property magic
  - Example: Modeling the social graph
Moderator

http://tinyurl.com/complextalk
List properties
What is a list property?

- Property in the Datastore that has multiple values
- An ordered list that maintains its order
- Queried with an equals filter
  - Any value in the list may cause a match
  - Sort order not useful without a composite index
- As easy as:

```python
class Favorites(db.Model):
  colors = db.StringListProperty()
  username = db.StringProperty()

fav.colors = ["red", "blue", "green"]
```
Why use list properties?

- Densely pack information
  - Track lists of related items
  - Use multiple parallel properties for storing "tuple"-like data
- Easy: compare to this one-to-many query

```python
class FavoriteColors(db.Model):
    color = db.StringProperty()
    username = db.StringProperty()

db.GqlQuery(
    "SELECT * FROM FavoriteColors "
    "WHERE username = :1", ...
)"
Why use list properties? (2)

- Great for answering set-membership questions
  - e.g., Which users like the color yellow?

```
results = db.GqlQuery(
    "SELECT * FROM FavoriteColors 
    "WHERE color = 'yellow'")

users = [r.username for r in results]
```

- This query matches any value of "yellow" in users' lists of favorite colors across all FavoriteColors entities.
Why use list properties? (3)

- Avoids storage overhead
  - Each list item only has an index entry
  - No keys for entities in a one-to-many relationship
  - No entry in the "by-kind" index
- Ultimately: Saves you a ton of storage space
- Simpler to understand than a normalized schema
  - It's just a list!
Why use list properties? (4)

- Gotchas
  - Uses more CPU for serializing/deserializing the entity when it's accessed
  - Works with sort orders *only* if querying a single list property; otherwise indexes "explode"
Concrete example: Microblogging

- Essentially: Publish/subscribe, broadcast/multicast
  - Users send a single message that goes to many other users

- It's a great example of fan-out
  - One user action causes a lot of work
  - Work leaves large amount of data to surface
  - Fan-out is hard!
Concrete example: Microblogging (2)

- Fan-out can be inefficient, require duplicate data
  - Send a copy of a message to N users
Concrete example: Microblogging (3)

- Efficient fan-out should not duplicate any data
  - Only overhead is cost of indexes
Concrete example: Microblogging, with RDBMS

**Users table**

<table>
<thead>
<tr>
<th>User ID</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
</tr>
</tbody>
</table>

**Messages table**

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>Hi there....</td>
</tr>
<tr>
<td>57</td>
<td>Echo...</td>
</tr>
</tbody>
</table>

**UsersMessages table**

<table>
<thead>
<tr>
<th>User ID</th>
<th>Message ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>1</td>
<td>82</td>
</tr>
</tbody>
</table>
Concrete example: Microblogging, with RDBMS (2)

- SQL query to find messages for user 'X' would be:

```
SELECT * FROM Messages
INNER JOIN UserMessages USING (message_id)
WHERE UserMessages.user_id = 'X';
```

- No joins on App Engine-- how do we do this?
  - List properties to the rescue!
Concrete example: Microblogging, with App Engine

```python
class Message(db.Model):
    sender = db.StringProperty()
    receivers = db.StringListProperty()
    body = db.TextProperty()

results = db.GqlQuery(
    "SELECT * FROM Message "
    "WHERE receivers = :1", me)
```

- That's it!
  - This is how Jaiku works
Concrete example: Microblogging, with JDO

@PersistenceCapable(
    identityType=IdentityType.APPLICATION)
public class Message {
    @PrimaryKey
    @Persistent(valueStrategy=
        IdGeneratorStrategy.IDENTITY)
    Long id;

    @Persistent String sender;
    @Persistent Text body;
    @Persistent List<String> receivers;
}
Concrete example: Microblogging, with JDO (2)

```java
pm = PMF.get().getPersistenceManager();
Query query = pm.newQuery(Message.class);
query.setFilter("receivers == 'foo'");
List<Message> results =
  (List<Message>) query.execute();
```
Concrete example: Microblogging Demo
List property performance

- Index writes are done in parallel on Bigtable
  - Fast-- e.g., update a list property of 1000 items with 1000 row writes simultaneously!
  - Scales linearly with number of items
  - Limited to 5000 indexed properties per entity

- Storage cost same as traditional RDBMS
  - RDBMS: User key + message key
  - Datastore: Entity key + list property value
List property performance (2)

- Downside: Serialization overhead
- Writes must package all list values into one serialized protocol buffer
  - OK because writes are relatively infrequent
- But queries must unpackage all result entities
  - When list size > ~100, reads are too expensive!
  - Slow in wall-clock time
  - Costs too much CPU
Improving list property performance

- Querying for messages should only return the message information
  - We don't care about the list properties after querying; this is why inner joins are useful

- What if we could selectively *skip* certain properties when querying?
  - Would avoid the serialization cost
  - Ideally, it would be great to do this in GQL:
    - SELECT foo, bar FROM MyModel ...
    - But this is not available...
Solution-- Relation index entities

- Split the message into two entities
  - Message model contains the info we care about
  - MessageIndex has only relationships for querying

```python
class Message(db.Model):
    sender = db.StringProperty()
    body = db.TextProperty()

class MessageIndex(db.Model):
    receivers = db.StringListProperty()
```
Solution-- Relation index entities (2)

- Put entities in the same entity group for transactions
Solution-- Relation index entities (3)

- Do a key-only query to fetch the MessageIndexes
- Transform returned keys to retrieve parent entity
- Fetch Message entities in batch

```python
indexes = db.GqlQuery(
    "SELECT __key__ FROM MessageIndex "
    "WHERE receivers = :1", me)

keys = [k.parent() for k in indexes]
messages = db.get(keys)
```

- Our Datastore works like this under the covers
Concrete example: Microblogging Demo (2)
Relation index entities: Conclusion

- Performance is much better
  - Writes same cost, reads ~10x faster/cheaper
- Best of both worlds with list properties:
  - Low storage cost, low CPU cost

- Even better: Scalable indexes
  - Need more indexes? Write multiple relation index entities per Message
  - Add indexes in the background (with Task Queue)
  - Solution for the million-fan-out problem
  - No need for schema migration!
Relation index entities: Conclusion (2)

- Scalable indexes
Merge-join
What is merge-join?

- People say we don't support joins -- not totally true!
  - We do not support natural, inner, or outer joins
- **We do** support "merge-join" queries
  - A type of self-join query; join a table with itself
  - Combine many equality tests into a single query
  - Determines Venn-diagram-like overlaps in sets

- Example

![Venn diagram example showing the intersection of Spots, Horns, and 4 legs leading to Cows.](image-url)
Why use merge-join?

- Great for exploring your data
  - Practical limit of equality tests is high (10+ filters)
- No need to build indexes in advance
  - Ad-hoc queries
  - Reduces cost

- Provides advanced functionality
  - Example query in Gmail: Various labels, read/unread, month/year/day, number of replies, recipients, etc
Example merge-join

class Animal(db.Model):
    has = db.StringListProperty()
    color = db.StringProperty()
    legs = db.IntegerProperty()

results = db.GqlQuery("""SELECT * FROM Animal WHERE color = 'spots' AND has = 'horns' AND legs = 4""")
How does merge-join work?

- Not available in raw Bigtable
  - Similar optimizations in other DB systems

- All property indexes are stored in sorted order
- Datastore does a merge-sort at runtime
- Uses a "zig-zag" algorithm to efficiently join tables
  - Scan a single Bigtable index in parallel
How does merge-join work?

- Not available in raw Bigtable
  - Similar optimizations in other DB systems

- All property indexes are stored in sorted order
- Datastore does a merge-sort at runtime
- Uses a "zig-zag" algorithm to efficiently join tables
  - Scan a single Bigtable index in parallel

*Is this a Google Interview? :(*
Example merge-join

<table>
<thead>
<tr>
<th>Row key</th>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red,key=ant</td>
<td>legs=2,key=falcon</td>
</tr>
<tr>
<td>color=spots,key=bear</td>
<td>legs=2,key=pigeon</td>
</tr>
<tr>
<td>color=spots,key=cow</td>
<td>legs=4,key=cat</td>
</tr>
<tr>
<td>color=white,key=dog</td>
<td>legs=4,key=cow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>has=hair,key=cat</td>
</tr>
<tr>
<td>has=horns,key=cow</td>
</tr>
<tr>
<td>has=jaws,key=lion</td>
</tr>
<tr>
<td>has=jaws,key=shark</td>
</tr>
</tbody>
</table>

(Tables represent property indexes)
Example merge-join

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red, key=ant</td>
</tr>
<tr>
<td>color=spots, key=bear</td>
</tr>
<tr>
<td>color=spots, key=cow</td>
</tr>
<tr>
<td>color=white, key=dog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>legs=2, key=falcon</td>
</tr>
<tr>
<td>legs=2, key=pigeon</td>
</tr>
<tr>
<td>legs=4, key=cat</td>
</tr>
<tr>
<td>legs=4, key=cow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>has=hair, key=cat</td>
</tr>
<tr>
<td>has=horns, key=cow</td>
</tr>
<tr>
<td>has=jaws, key=lion</td>
</tr>
<tr>
<td>has=jaws, key=shark</td>
</tr>
</tbody>
</table>

(Tables represent property indexes)
Example merge-join

<table>
<thead>
<tr>
<th>Row key</th>
<th>Row key</th>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red,key=ant</td>
<td>legs=2,key=falcon</td>
<td>legs=2,key=falcon</td>
</tr>
<tr>
<td>color=spots,key=bear</td>
<td>legs=2,key=pigeon</td>
<td>legs=4,key=cat</td>
</tr>
<tr>
<td>color=spots,key=cow</td>
<td>has=jaws,key=lion</td>
<td>has=jaws,key=shark</td>
</tr>
<tr>
<td>color=white,key=dog</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Tables represent property indexes)
Example merge-join

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red,key=ant</td>
</tr>
<tr>
<td>color=spots,key=bear</td>
</tr>
<tr>
<td>color=spots,key=cow</td>
</tr>
<tr>
<td>color=white,key=dog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>legs=2,key=falcon</td>
</tr>
<tr>
<td>legs=2,key=pigeon</td>
</tr>
<tr>
<td>legs=4,key=cat</td>
</tr>
<tr>
<td>legs=4,key=cow</td>
</tr>
</tbody>
</table>

(Tables represent property indexes)
Example merge-join

Zig!

1

Row key
color=red,key=ant
color=spots,key=bear
color=spots,key=cow
color=white,key=dog

Row key
legs=2,key=falcon
legs=2,key=pigeon
legs=4,key=cat
legs=4,key=cow

2

Row key
has=hair,key=cat
has=horns,key=cow
has=jaws,key=lion
has=jaws,key=shark

(Tables represent property indexes)
Example merge-join

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red,key=ant</td>
</tr>
<tr>
<td>color=spots,key=bear</td>
</tr>
<tr>
<td>color=spots,key=cow</td>
</tr>
<tr>
<td>color=white,key=dog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>legs=2,key=falcon</td>
</tr>
<tr>
<td>legs=2,key=pigeon</td>
</tr>
<tr>
<td>legs=4,key=cat</td>
</tr>
<tr>
<td>legs=4,key=cow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>has=hair,key=cat</td>
</tr>
<tr>
<td>has=horns,key=cow</td>
</tr>
<tr>
<td>has=jaws,key=lion</td>
</tr>
<tr>
<td>has=jaws,key=shark</td>
</tr>
</tbody>
</table>

(Tables represent property indexes)
Example merge-join

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red,key=ant</td>
</tr>
<tr>
<td>color=spots,key=bear</td>
</tr>
<tr>
<td>color=spots,key=cow</td>
</tr>
<tr>
<td>color=white,key=dog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>legs=2,key=falcon</td>
</tr>
<tr>
<td>legs=2,key=pigeon</td>
</tr>
<tr>
<td>legs=4,key=cat</td>
</tr>
<tr>
<td>legs=4,key=cow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>has=hair,key=cat</td>
</tr>
<tr>
<td>has=horns,key=cow</td>
</tr>
<tr>
<td>has=jaws,key=lion</td>
</tr>
<tr>
<td>has=jaws,key=shark</td>
</tr>
</tbody>
</table>

*(Tables represent property indexes)*
**Example merge-join**

<table>
<thead>
<tr>
<th>Row key</th>
<th>Row key</th>
</tr>
</thead>
<tbody>
<tr>
<td>color=red,key=ant</td>
<td>legs=2,key=falcon</td>
</tr>
<tr>
<td>color=spots,key=bear</td>
<td>legs=2,key=pigeon</td>
</tr>
<tr>
<td><strong>color=spots,key=cow</strong></td>
<td><strong>legs=4,key=cat</strong></td>
</tr>
<tr>
<td>color=white,key=dog</td>
<td><strong>legs=4,key=cow</strong></td>
</tr>
</tbody>
</table>

*Tables represent property indexes*
Essentially: Users have a profile and a set of friends
  • Use merge-join on list properties-- magic!
• Answer queries about relationships
  • Who are my friends?
  • Who are my friends in location L?
  • Which friends do I have in common with person P?
  • Which friends do I have in common with person P in location L?
• For simplicity, this example assumes all relationships are two-way
  • Concept also works for directed acyclic graphs
Concrete example: Social graph (2)

- **New York**: Bob -> Mel
- **San Francisco**: Stu -> Willie
- **Chicago**: Lenny -> Carl

Lines are friendships
Concrete example: Social graph (2)
Concrete example: Social graph (2)

- **New York**
  - Bob
  - Mel

- **San Francisco**
  - Stu
  - Willie

- **Chicago**
  - Lenny
  - Carl

**Mutual friend of Bob and Willie in San Francisco**
Concrete example: Social graph, with RDBMS

**Person table**

<table>
<thead>
<tr>
<th>User ID</th>
<th>Location</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>San Francisco</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>New York</td>
<td>...</td>
</tr>
</tbody>
</table>

**Friends table**

<table>
<thead>
<tr>
<th>UserA ID</th>
<th>UserB ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>5</td>
</tr>
<tr>
<td>57</td>
<td>1</td>
</tr>
</tbody>
</table>
Concrete example: Social graph, with RDBMS (2)

- SQL query to find friends of user 'X':

```sql
SELECT * FROM Users
INNER JOIN Friends
ON Users.user_id = Friends.user_b_id
WHERE Friends.user_a_id = 'X'

- To also filter by location, add:

```sql
AND Users.location = 'San Francisco'
```
Concrete example: Social graph, with RDBMS (3)

- SQL query to find friends common to 'X' and 'Y':

```sql
SELECT * FROM Users
INNER JOIN Friends f1, Friends f2
ON Users.user_id = f1.user_b_id AND Users.user_id = f2.user_b_id
WHERE f1.user_a_id = 'X' AND f2.user_a_id = 'Y' AND f1.user_b_id = f2.user_b_id
```

- No inner joins in App Engine, what now?
  - We do have merge-join; we can do self-joins!
Concrete example: Social graph, with App Engine

class Person(db.Model):
    location = db.StringProperty()
    friends = db.StringListProperty()

db.GqlQuery(
    """SELECT * FROM Person WHERE
    friends = :1 AND
    friends = :2 AND
    location = 'San Francisco'""",
    me, otherguy)

● That's it!
  ○ Add as many equality filters as you need
Concrete example: Social graph Demo
Merge-join performance

- Scales with number of filters and size of result set
  - Best for queries with fewer results (less than 100)
- Similar access performance as list properties
  - Same read/write speed
  - No extra storage overhead
  - Can avoid serialization with relation index entities
Merge-join performance (2)

- Gotchas
  - Watch out for pathological datasets!
    - Too many overlapping values = lots of zig-zagging
  - Doesn't work with composite indexes because of "exploding" index combinations
  - That means you can't apply sort orders!
    - Must sort in memory
Wrap-up
Wrap-up

- Use list properties and merge-join for many things
  - Fan-out
  - Geospatial info
  - Relationship graphs
  - "Fuzzy" values

- Think about how to convert your queries into "set membership" tests
- Compute membership at write time, enjoy fast reads!
Wrap-up (2)

- Demos available with source code
  - [http://pubsub-test.appspot.com](http://pubsub-test.appspot.com)
  - [http://dagpeople.appspot.com](http://dagpeople.appspot.com)

- More info on our site
  - [http://code.google.com/appengine](http://code.google.com/appengine)
Questions?