Google Developer Day2009

Building scalable, complex apps on App Engine

Fred Sauer
June 9, 2009

Based on original presentation by Brett Slatkin

Google Developer Day 2009

What we will cover today

- 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

List Properties

What is a list property?

- Property in the Datastore that has multiple values
- As easy as:

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

fav.colors = ["red", "green", "blue"]
```

- An ordered list
- Which 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)

How can we use list properties?

- Track lists of related items
- Use multiple parallel properties for storing "tuple"-like data

```
players.names  = ["joe", "jane", "john"]
players.scores = [1290,  54800,  360   ]
```

- Easy: compare to this one-to-many query:

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

db.gql("SELECT * FROM FavoriteColors "
       "WHERE username = :1", ...)
```

How can we use list properties? (2)

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

```
results = db.gql(
    "SELECT * FROM Favorites "
    "WHERE colors = 'yellow'")

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

- Great fan-out capability: cut across all your data
  - This query matches *any* value of "yellow" in *any* users' list of favorite colors across *all* Favorites entities.

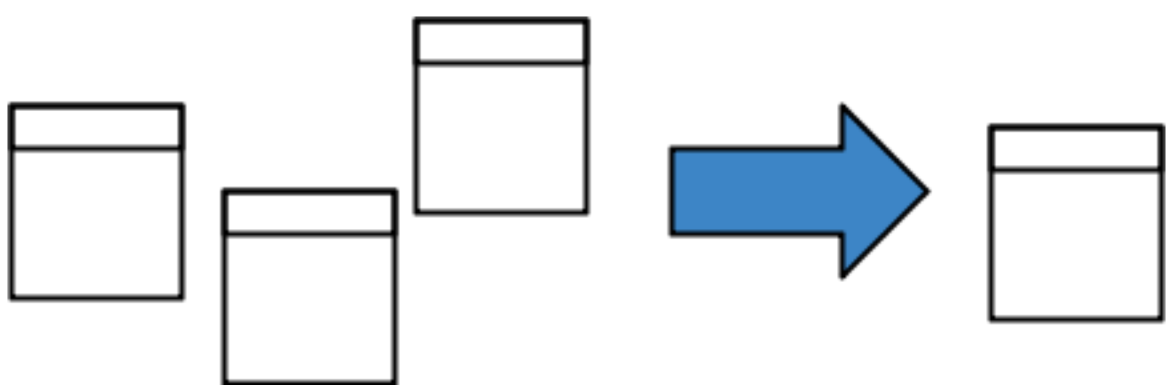
Why use list properties?

Avoids storage overhead:

- Each list item only has an index entry
- No entry in the "by-kind" index
- No key for entities in a one-to-many relationship

- Ultimately: Saves you a ton of storage space

- Simpler to understand than a normalized schema
  - It's just a list!



List property 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"

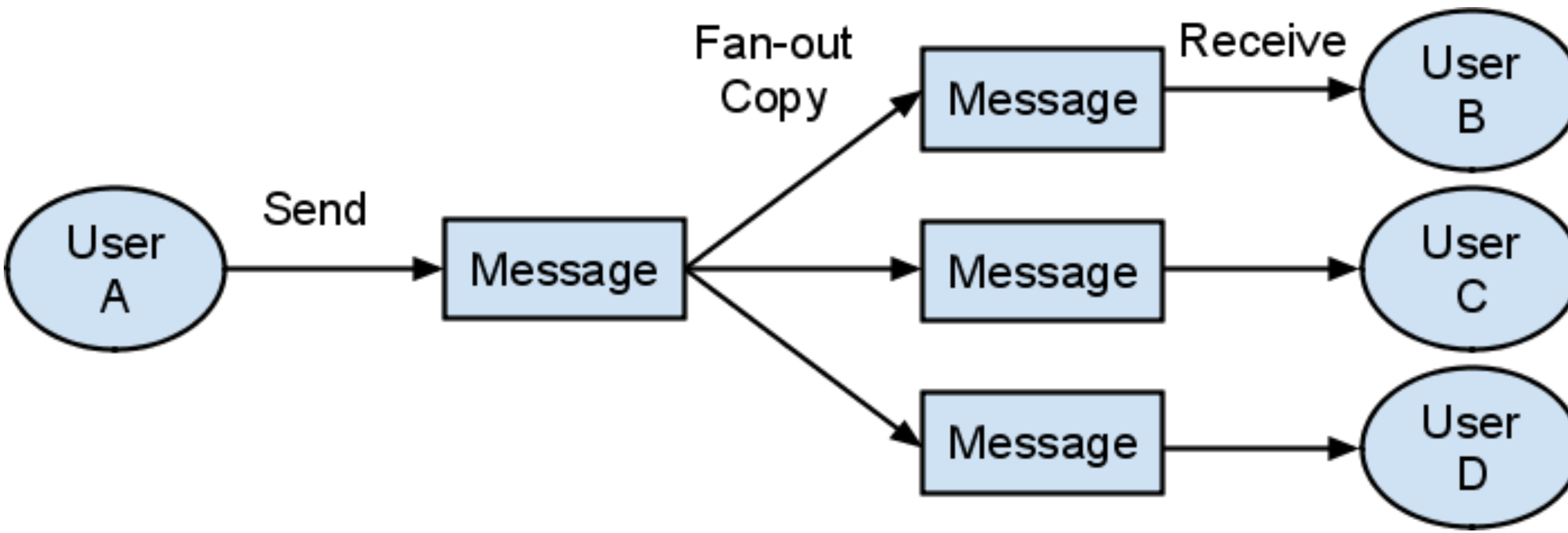
Concrete example: Microblogging

- 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!

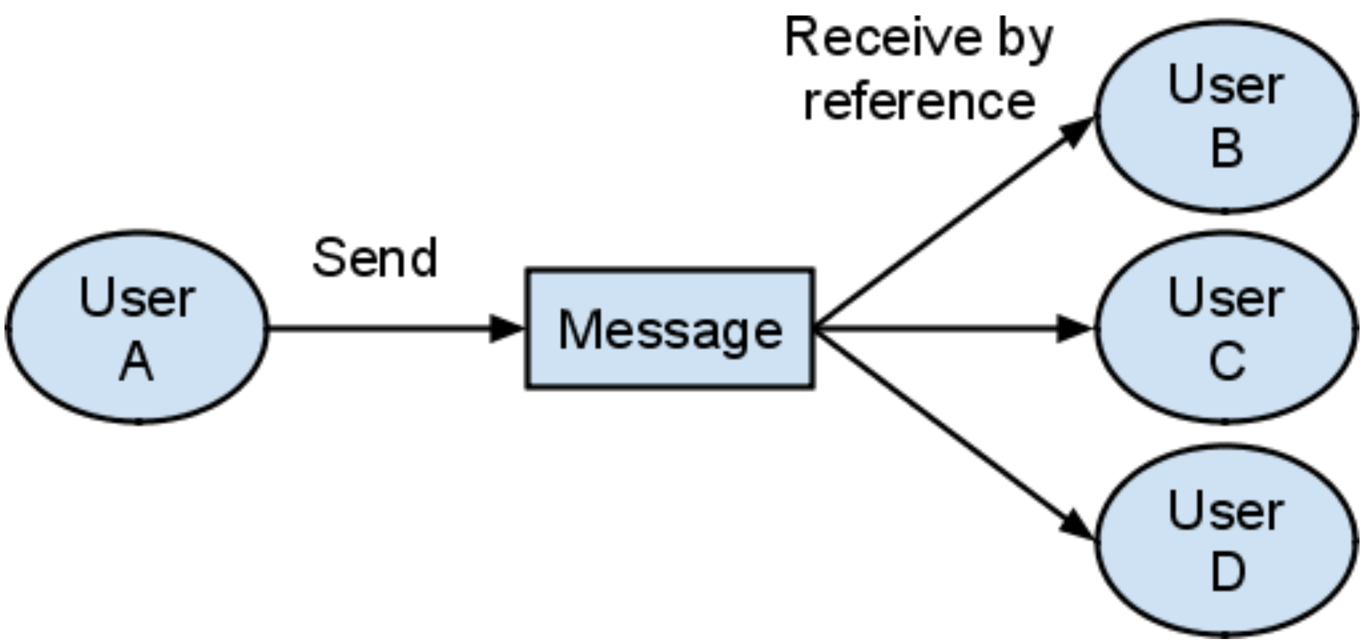
Concrete example: Microblogging (2)

- 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

User ID	Name
42	Ford
43	...

Messages table

Message ID	Body
56	Hi there....
57	Echo...

UsersMessages table

User ID	Message ID
42	56
42	82

Concrete example: Microblogging, with RDBMS (2)

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

```
SELECT * FROM Messages
INNER JOIN UserMessages USING (message_id)
WHERE UserMessages.user_id = 42;
```

- No joins on App Engine-- how do we do this?
  - List properties to the rescue!

Concrete example: Microblogging, with App Engine

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

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)

```
pm = PMF.get().getPersistenceManager();
Query query = pm.newQuery(Message.class);
query.setFilter("receivers == 'foo'");
List<Message> results =
    (List<Message>) query.execute();
```

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: List property value + Entity key

List property performance (2)

- Downside: Serialization overhead
  - Not to worry, there's a solution
- 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

*Protocol buffers:
http://code.google.com/p/protobuf/

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 we only have:

```
SELECT * FROM MyModel ...
```

Solution-- Relation Index Entities

Relation Index Entities

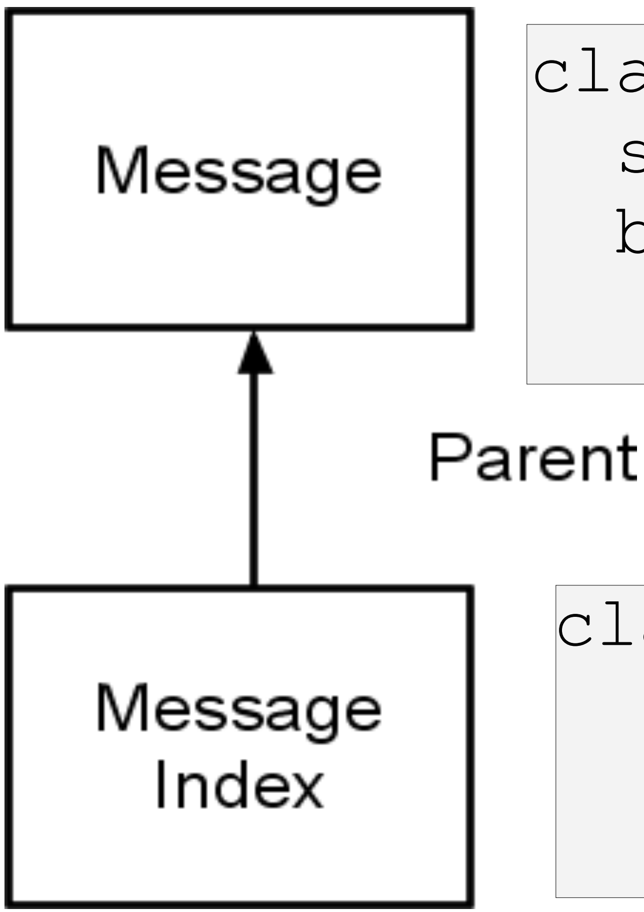
- Split the message into two entities
  - **Message** contains the info we care about
  - **MessageIndex** has only relationships for querying

```
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




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

```
class MessageIndex(db.Model):
  receivers =
      db.StringListProperty()
```

Solution-- Relation Index Entities (3)

- Do a key-only query to fetch the **MessageIndex**es

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

- Transform returned keys to retrieve parent entity

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

- Fetch **Message** entities in batch

```
messages = db.get(keys)
```

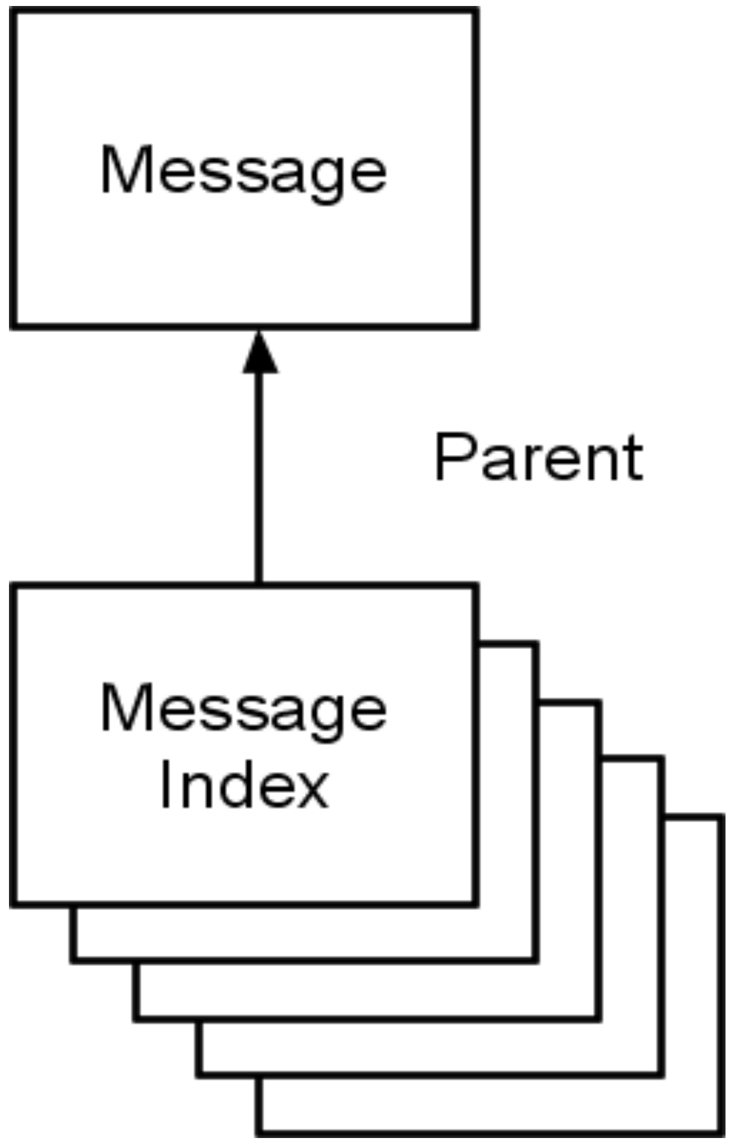
- Our Datastore works like this under the covers

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 **MessageIndex**es 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

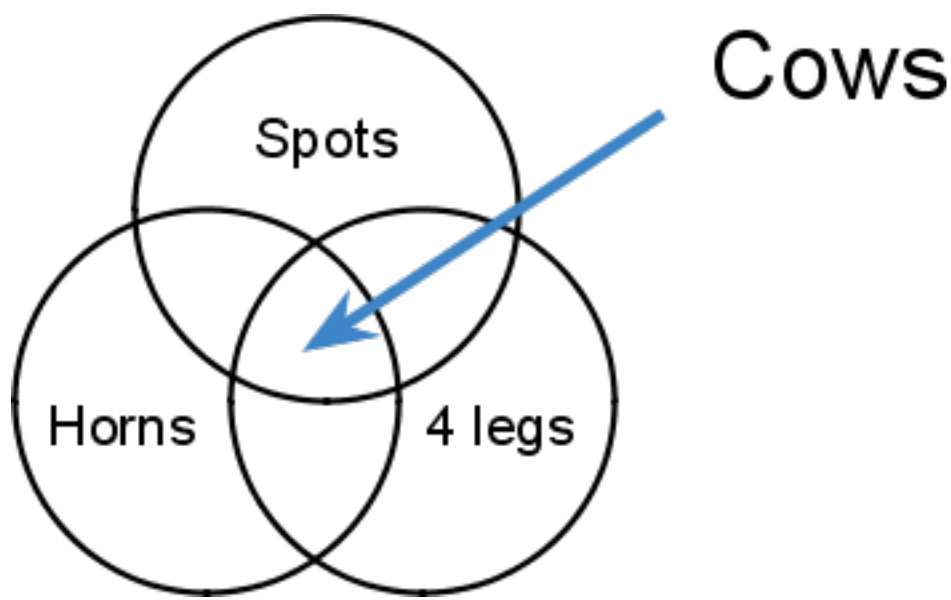


Merge-join

Google Developer Day2009

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:



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

Example merge-join

Row key
color=red,key=ant
color=spots,key=bear
color=spots,key=cow
color=white,key=dog
has=hair,key=cat
has=horns,key=cow
has=jaws,key=lion
has=jaws,key=shark
legs=2,key=falcon
legs=2,key=pigeon
legs=4,key=cat
legs=4,key=cow

Like everything in BigTable, the property index rows are sorted

(Tables represent property indexes)

Example merge-join

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

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

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

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

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

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

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

Zig!

(1: color=spots,key=cow)

(2: has=horns,key=cow)

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

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

1

Zig!

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

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

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

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

1

2

3

Zag!

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

1

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

3

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

Match!

2

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

(Tables represent property indexes)

Concrete example: Social graph

Essentially: Users have a profile and a set of friends

- Use merge-join on list properties-- ***magic!***

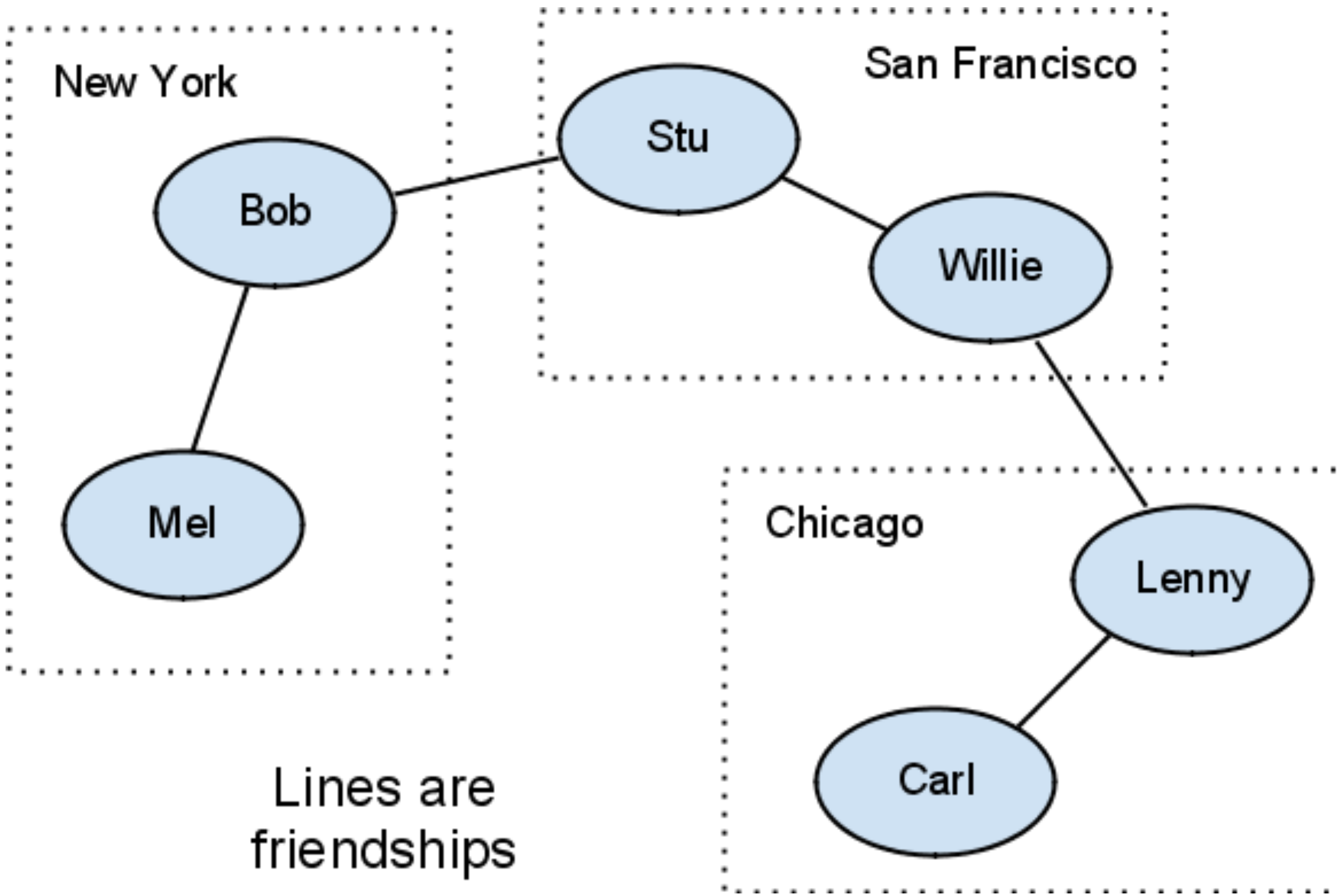
Answer queries about relationships

- 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

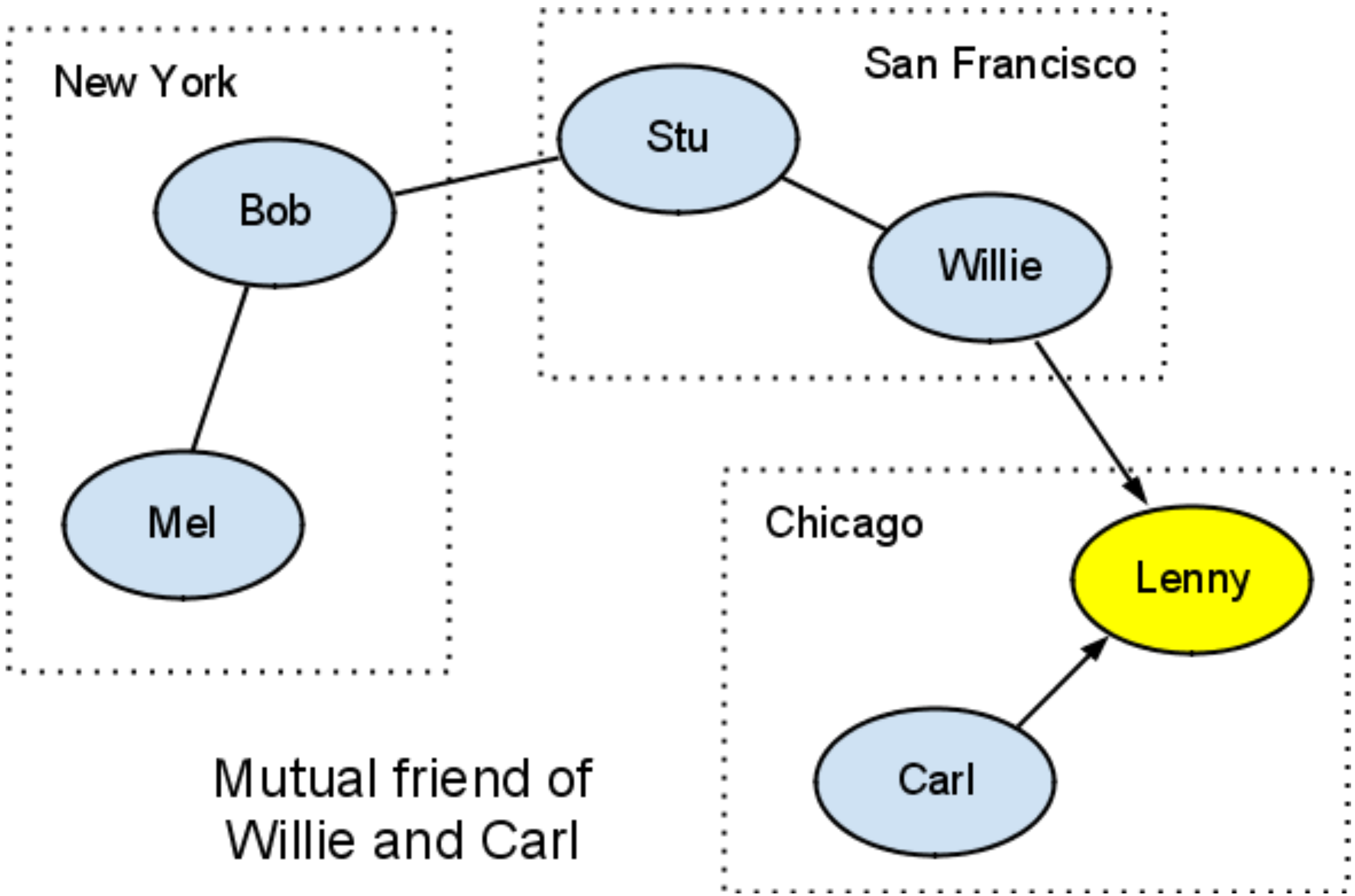
- Concept also works for directed acyclic graphs

Concrete example: Social graph (2)

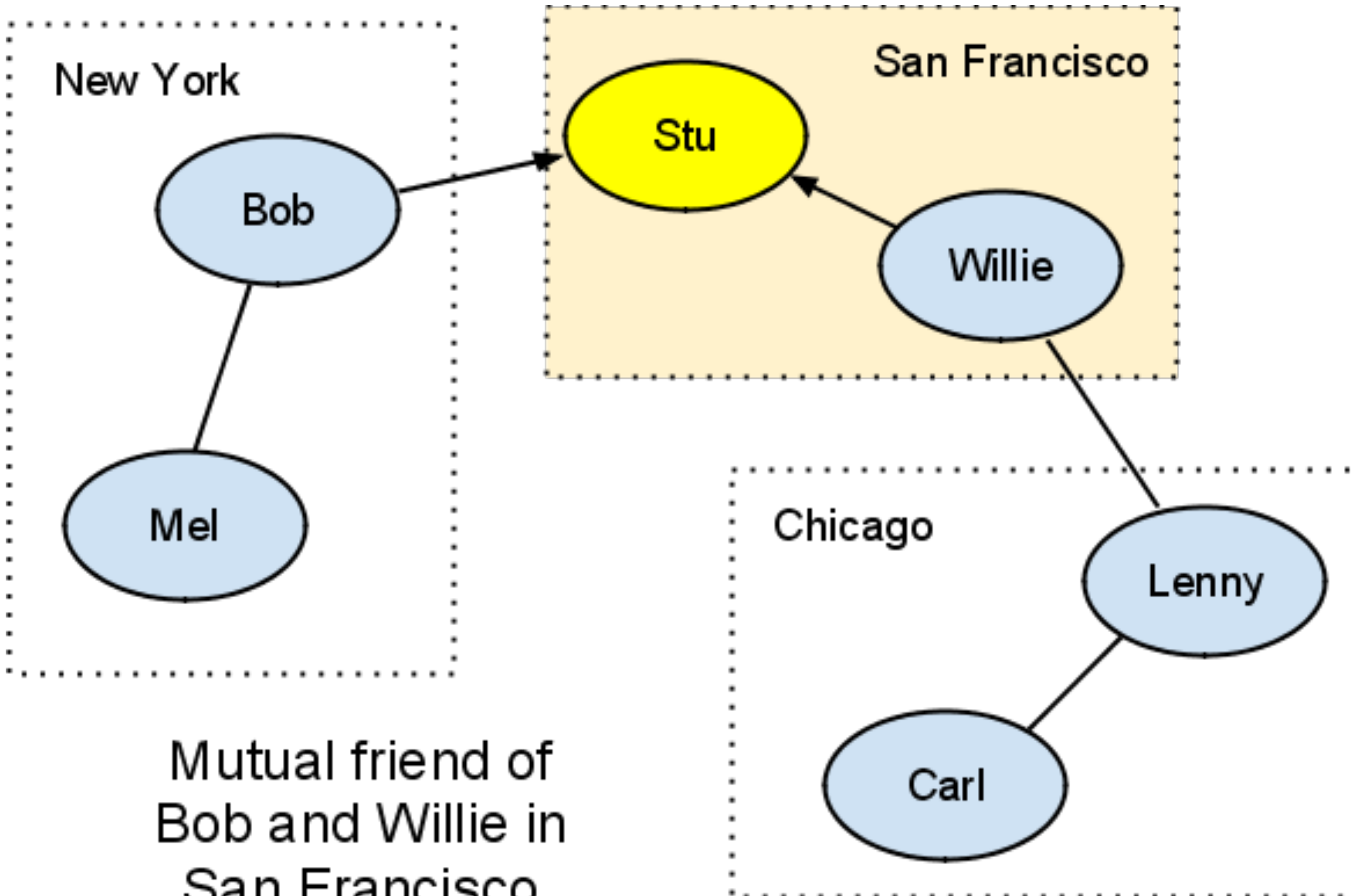


Google Developer Day2009

Concrete example: Social graph (2)



Concrete example: Social graph (2)



Mutual friend of
Bob and Willie in
San Francisco

Concrete example: Social graph, with RDBMS

Person table

User ID	Location	...
1	San Francisco	...
2	New York	...

Friends table

UserA ID	UserB ID
56	5
57	1

Concrete example: Social graph, with RDBMS (2)

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

```
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:

```
AND Users.location = 'San Francisco'
```

Concrete example: Social graph, with RDBMS (3)

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

```
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

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!

Thank You

Read more

http://code.google.com/appengine/

- Demos available with source code
  - http://pubsub-test.appspot.com/
  - http://dagpeople.appspot.com/

Contact info

Fred Sauer (twitter: @fredsa)Developer Advocate

fredsa@google.com

Questions

?

Google
Developer
Day2009