

Google App Engine 上でスケールするWebアプリを書く

Brett Slatkin May 28th, 2008

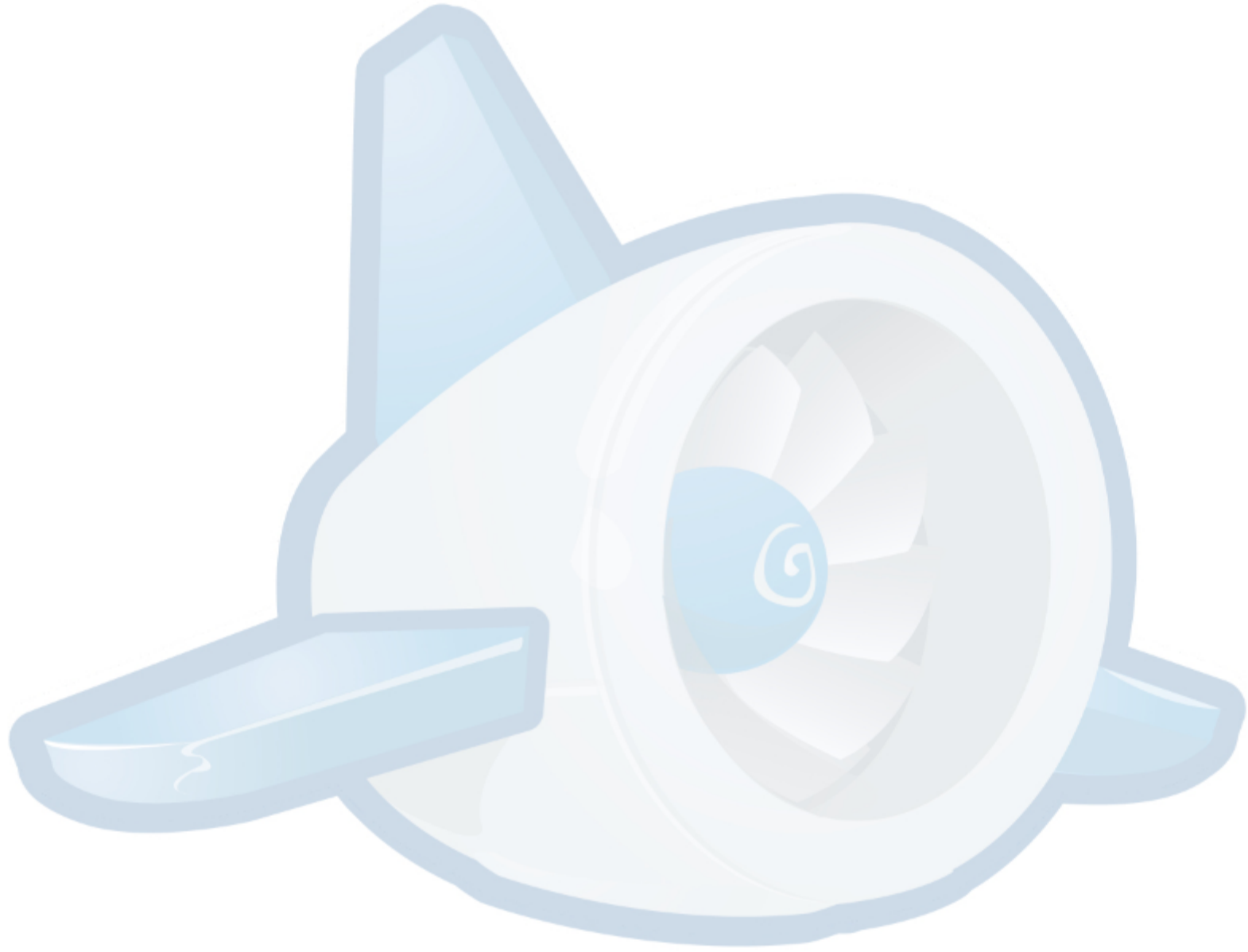
Takashi Matsuo 訳

アジェンダ

- Python ランタイムを有効に使おう
- 知っておくべき数字
- 大きなデータセットを扱うツール
- サンプルコード: 分散カウンター
- サンプルコード: ブログ



無駄な繰り返しをしない



無駄な繰り返しをしない

- リクエスト毎に Python module を読むと遅くなる
- main() を再利用して対応:

```
def main():  
    wsgiref.handlers.CGIHandler().run(my_app)  
if __name__ == "__main__":  
    main()
```

- **大きなモジュール** を遅延ロードすればワームアップコストを減らせる

```
def my_expensive_operation():  
    import big_module  
    big_module.do_work()
```

- 「読み込み済み」モジュールの利点を生かす

無駄な繰り返しをしない 2

大きなリザルトセットを避ける

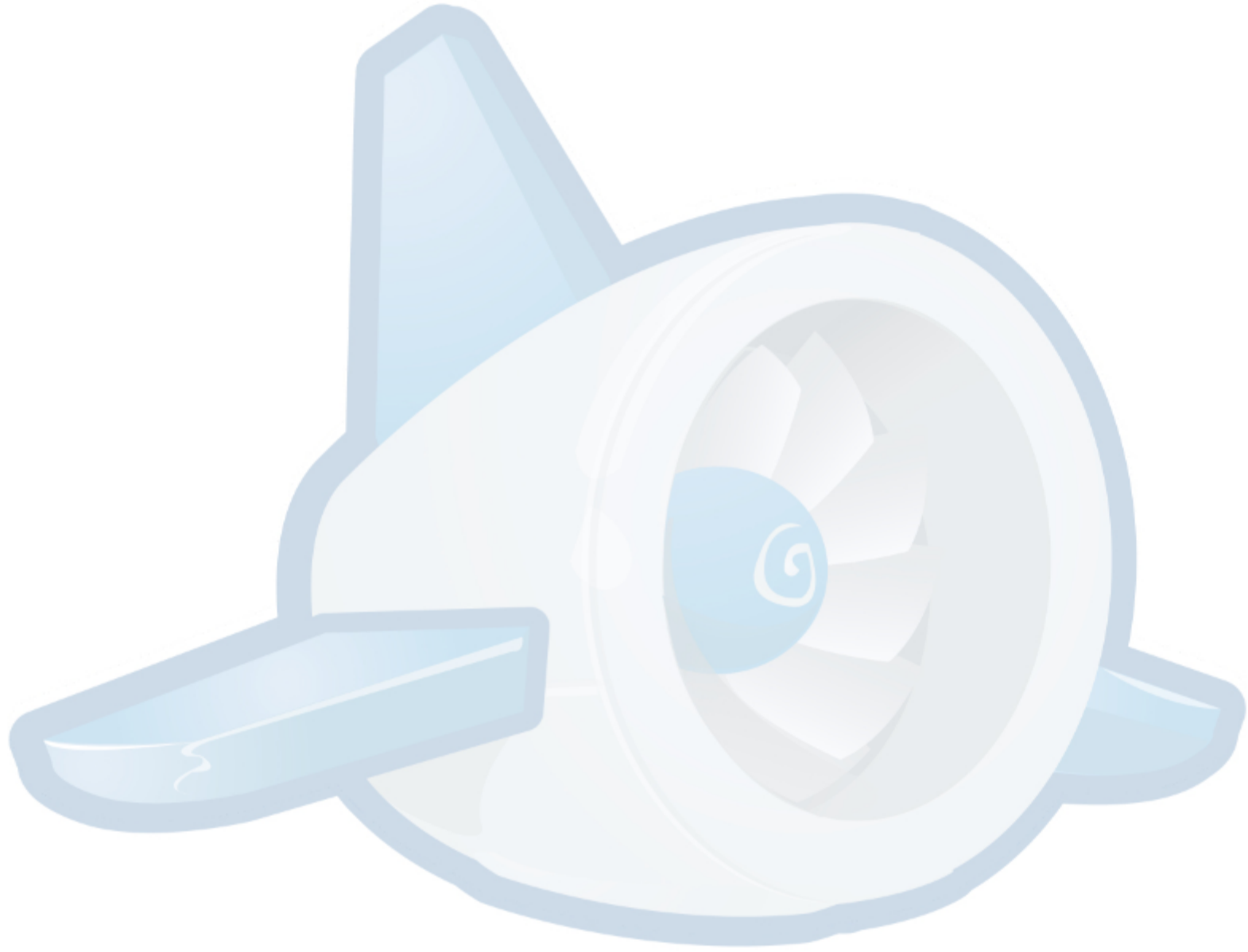
- メモリでソートやフィルタを行うのは遅い
- データストアにやらせよう

同じクエリーを繰り返さないようにする

- アプリケーションのトップページ等は全員に対して同じクエリーをする事が多い
- Incoherent caching
- 更新が少ないビューにはmemcache を使用

```
results = memcache.get('main_results')
if results is None:
    results = db.GqlQuery('...').fetch(10)
    memcache.add('main_results', results, 60)
```

知っておくべき数字



知っておくべき数字

書き込みは高コスト!

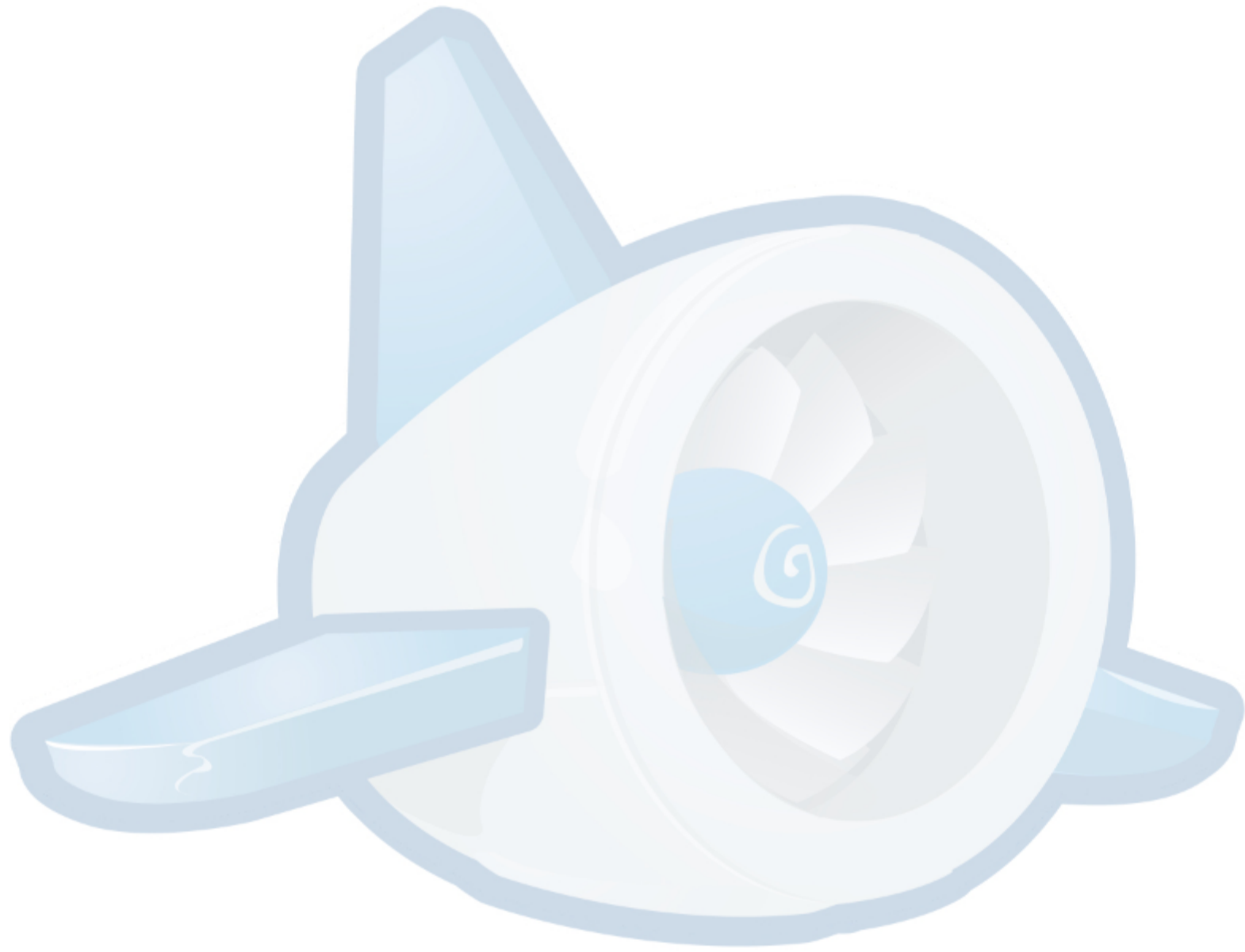
- データストアはランザクショナル:書き込みにはディスクアクセスが必要
 - ディスクアクセス、つまりディスクシーク
- 大雑把に:ディスクシークには 10ms かかる
- 単純計算:
 - $1s / 10ms = \text{最大 } 100 \text{ seeks/sec}$
- 下記に依存
 - データの大きさと形式
 - Doing work in batches (batch puts and gets)

知っておくべき数字 2

読み込みは低コスト!

- 読み込みにはランザクションは必要無い。整合性さえ取れていれば良い。
- はじめデータはディスクから読まれるが、容易にキャッシュできる。
- 後に続く読み込みは全てメモリから直接データを取得
- 大雑把に:メモリから 1MB 読むのに 250usec
- 単純計算:
 - $1s / 250usec = \text{最大 } 4GB / sec$
 - 1MB のエンティティなら、4000回取得 / 秒

データを保存する

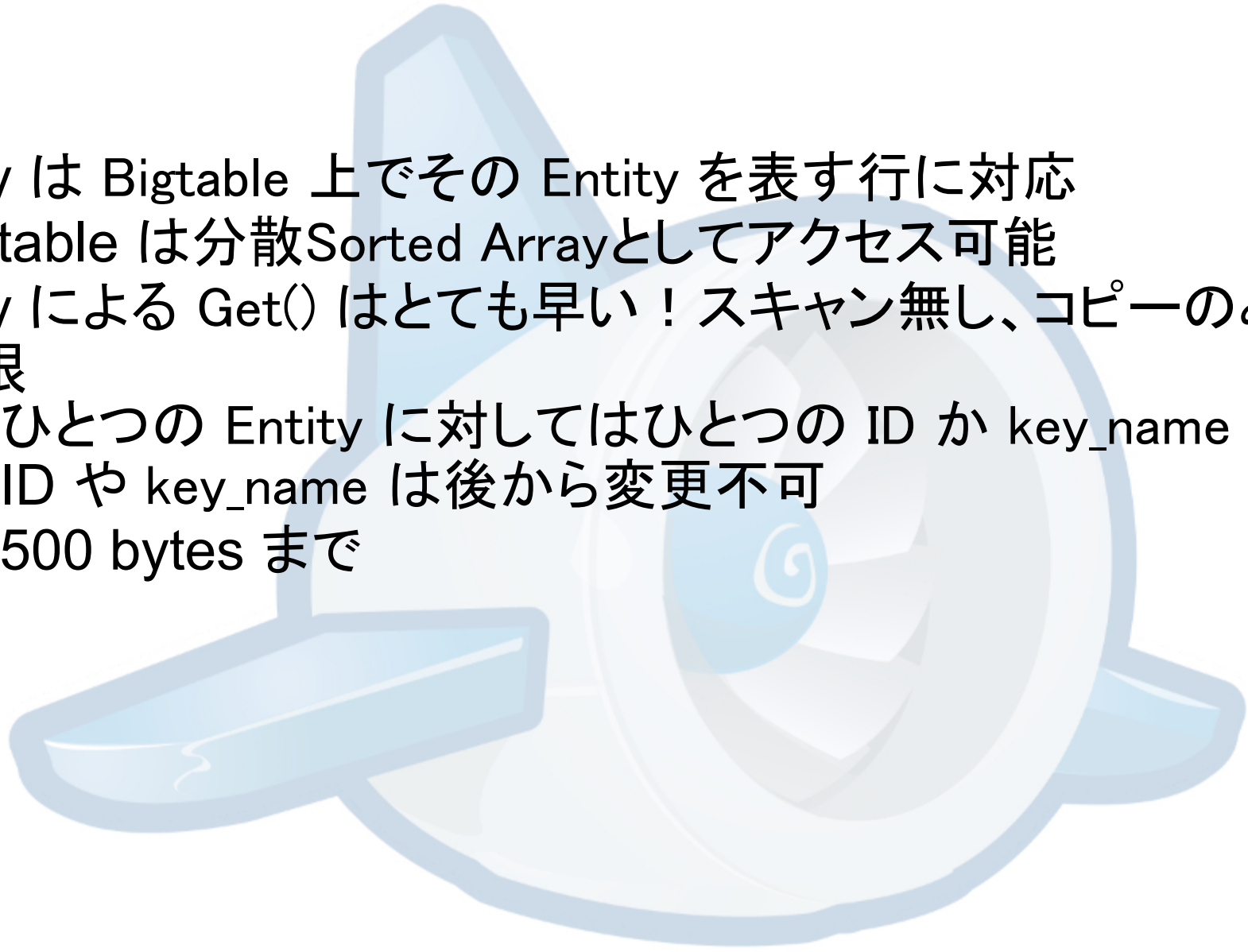


データを保存する: Entities

- App Engine における基本ストレージ
- プロパティの名前/値 のペアで構成される
- 殆どのプロパティに対してインデックスが作られて、高速に検索
- 他の大きなプロパティに対してはインデックスが作られない (Blobs, Text等)
- RDB とは違う。オブジェクトの保存と考える。
 - Kinds はクラスのようなもの
 - Entities はインスタンスのようなもの
- Entities 同士のリレーション
 - Reference プロパティを使う
 - One to Many も Many to Many も可能

データを保存する: Keys

- Key は Bigtable 上でその Entity を表す行に対応
- Bigtable は分散Sorted Arrayとしてアクセス可能
- Key による Get() はとても早い！ スキャン無し、コピーのみ
- 制限
 - ひとつの Entity に対してはひとつの ID か key_name だけ
 - ID や key_name は後から変更不可
 - 500 bytes まで



データを保存する: Transactions

- ACID Transactions
 - Atomicity, Consistency, Isolation, Durability
- Transaction 内ではクエリー不可
- Get() と Put() を使ったトランザクショナルな読み書きのみ
- 一般的な方法
 - クエリーを使って必要なものを取得
 - Get() と Put() を使ってトランザクションを行う
- どのようにクエリーの結果に整合性を持たせるか

データを保存する: Entity Groups

密接に関連した Entities はひとつの Entity Group に入れられる

- お互い論理的・物理的に近い場所に保存される

トランザクション性を理解する

- RDBMS の場合: 行ロックとテーブルロック
- Datastore の場合: 単独の Entity Group にまたがるトランザクション
- グループ内の Entity をひとつロックすると Group 全部ロックされる
- (トランザクション内では)書き込みは serialize される
- 伝統的なロックとは違い、書き込みを行うプロセスは並行して書き込みを完了させようと試みる

データを保存する: Entity Groups2

階層構造

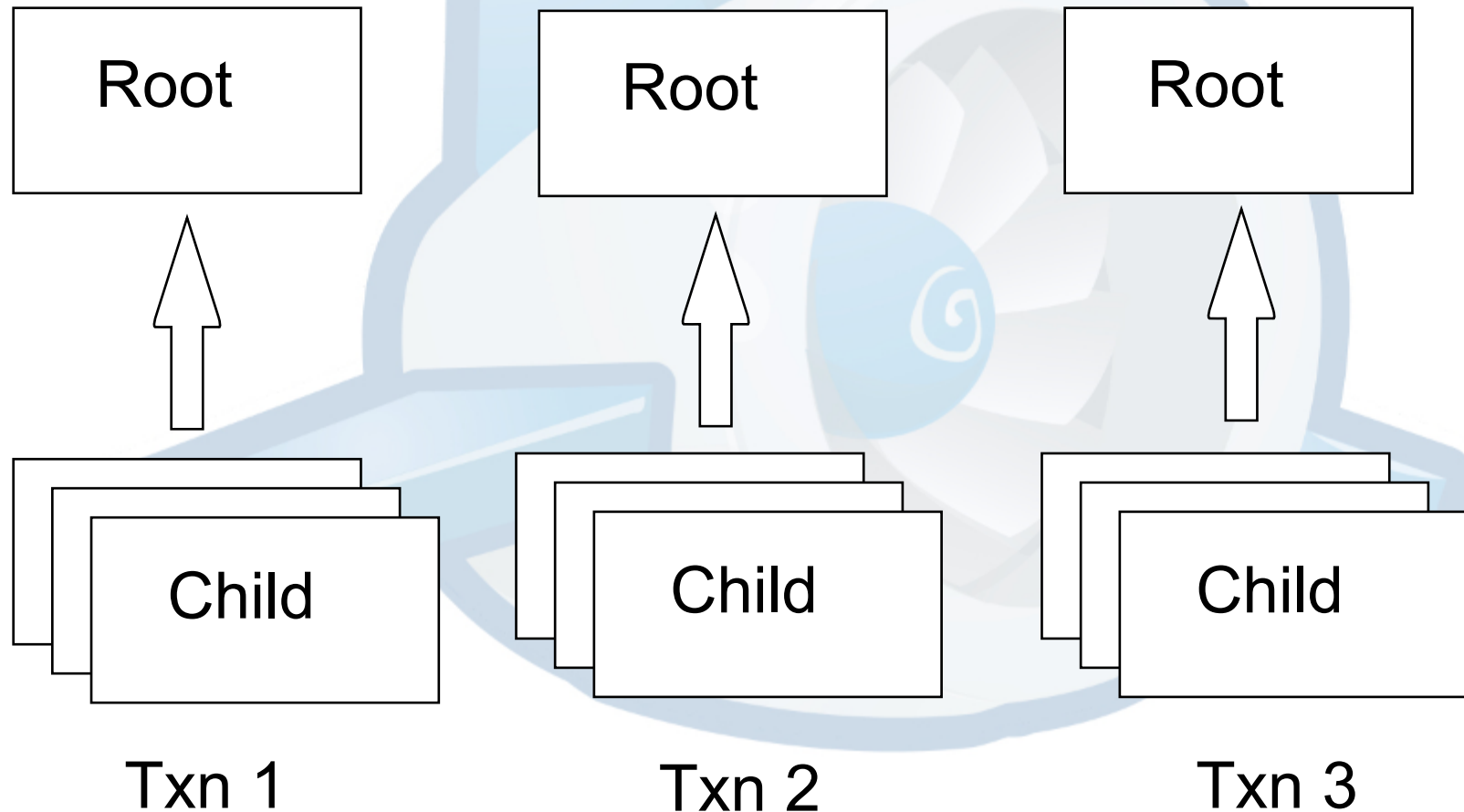
- どの Entity も parent を持てる
- 階層の root が Entity group を定義する
- 子の Entity 階層はどんどん深くできる
 - 注意: 全ての子に対して書き込みが serialize される

Datastore は広くスケールする

- Entity group 毎に serialize された書き込みがなされる
- 並行していくつでも Entity group を同時に使える
- たくさんの独立した階層データだと考えられる

データを保存する: Entity Groups3

別の Entity group は並行してトランザクション処理が可能



データを保存する: Entity Groups4

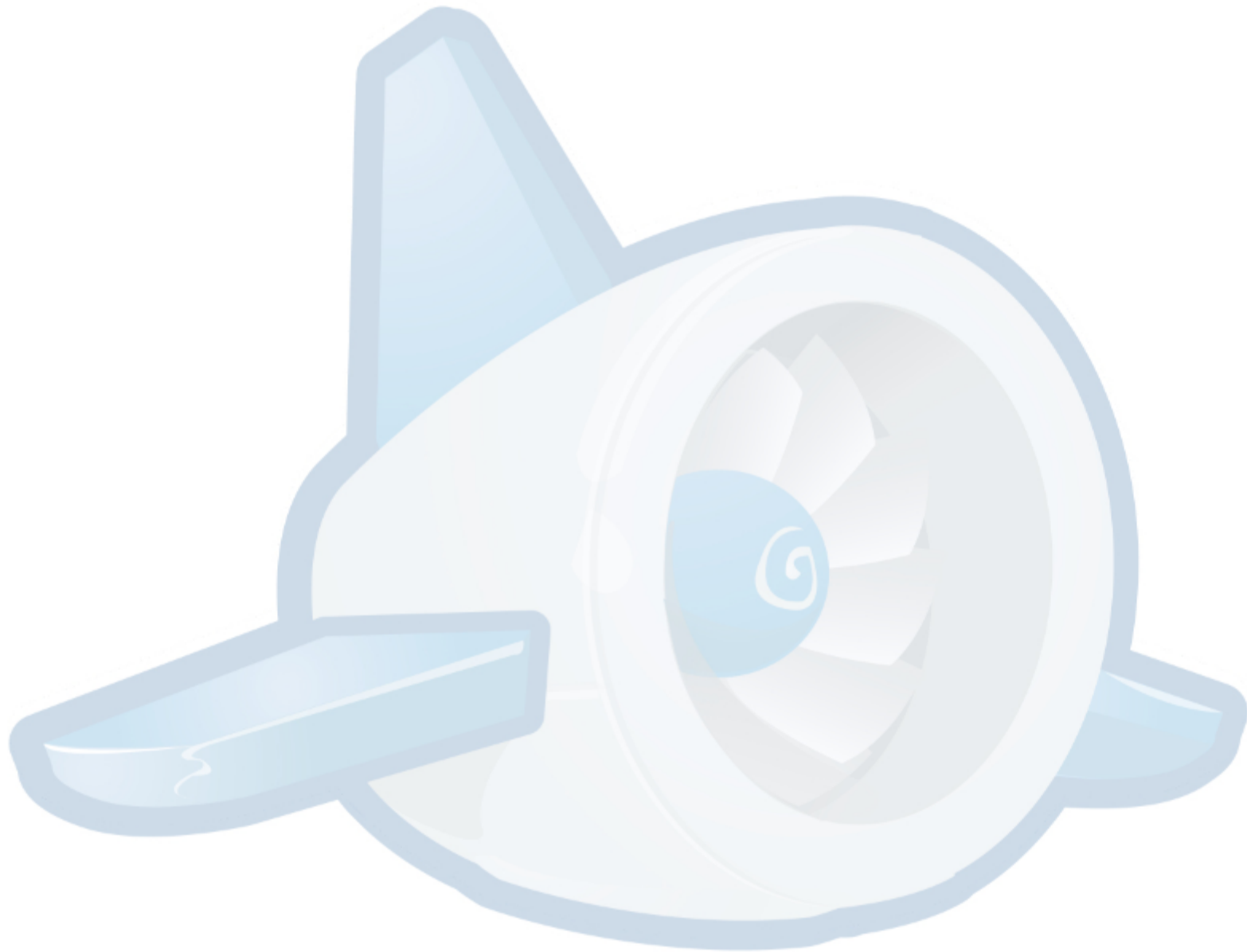
落とし穴

- 大きな Entity group = high contention = トランザクションの失敗が増える
- 書き込みのスループットを考えないとダメ
 - データの使用パターンに合うようにデータ構造を作る

良いニュースも

- Entity group をまたがったクエリーは serialize されない
- Entity group 全体として矛盾の無い View
 - 部分的なコミットは見えない
 - group 内の全ての Entity は最後にコミットされたバージョン

カウンター



カウンター

Model.count() を使う

- Bigtable はデザイン上総数を知らないのよ
- $O(N)$; $O(1)$ ではない！ 全ての行を舐める必要がある

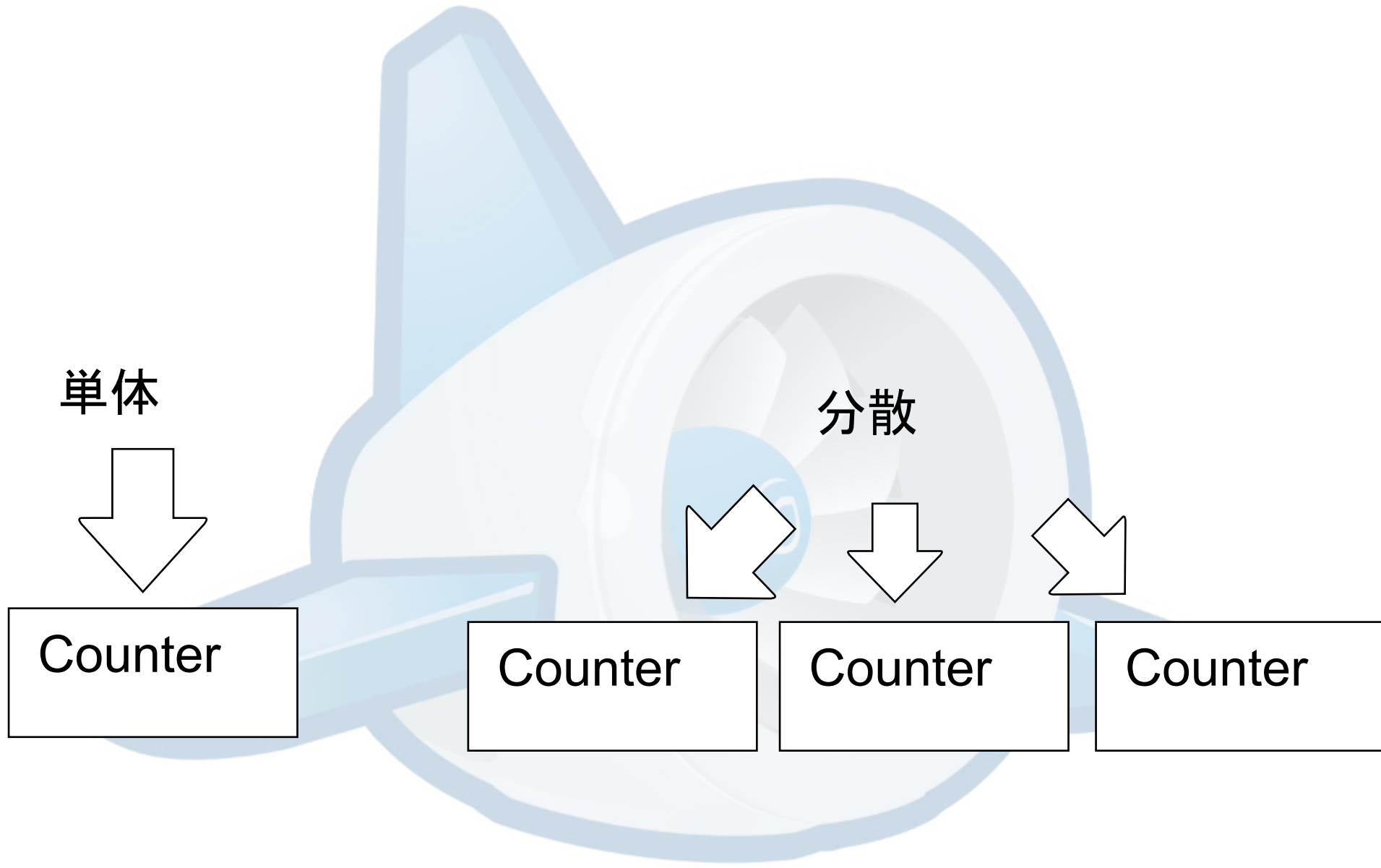
count プロパティを持った Entity を使う

```
class Counter(db.Model):
```

```
    count = db.IntegerProperty()
```

- ひんぱんにアップデートされると \rightarrow high contention
- トランザクション書き込みは serialize されてとても遅い
- これは分散システムの基本的な制約

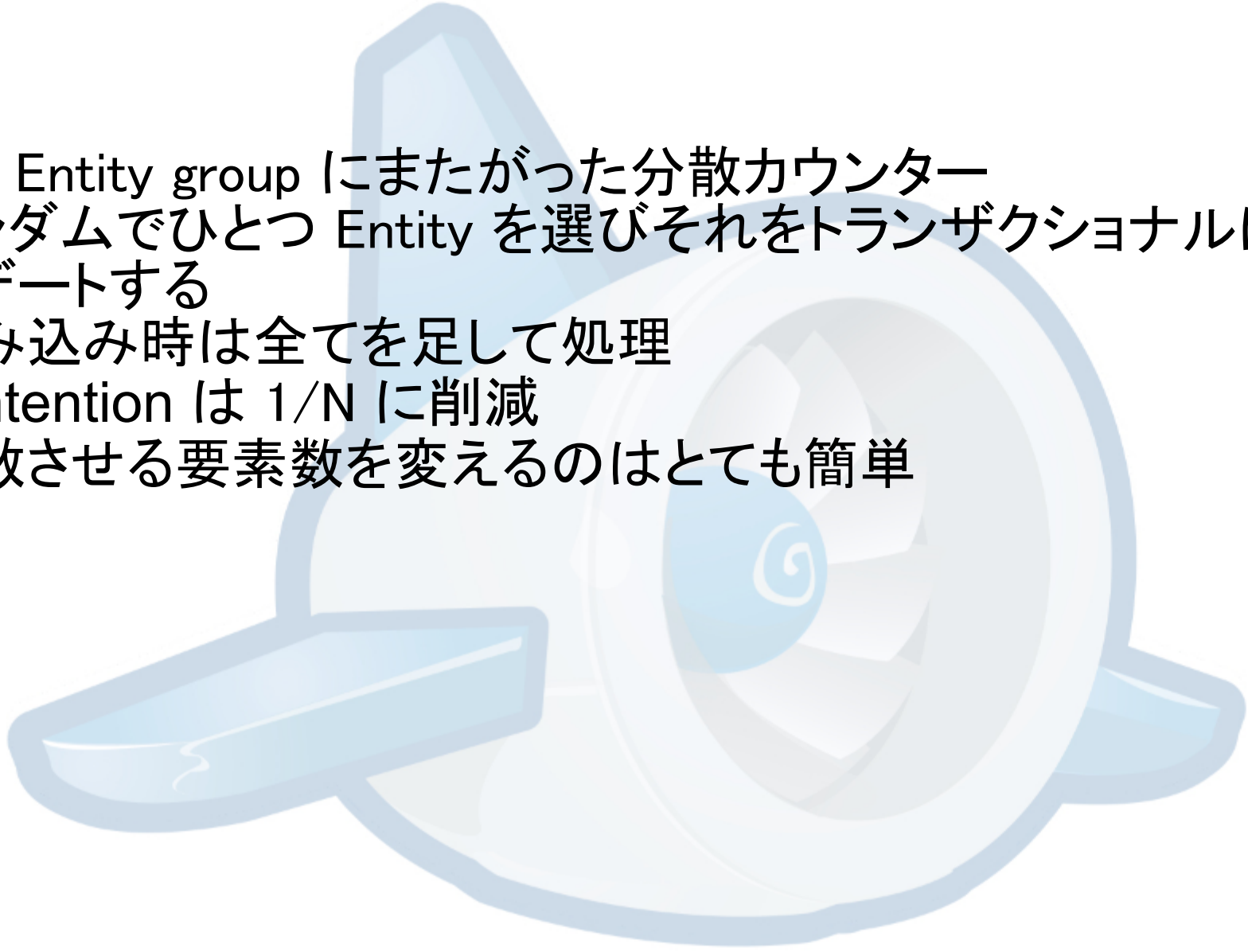
カウンター: ビフォーアフター



カウンター: 分散カウンター

複数の Entity group にまたがった分散カウンター

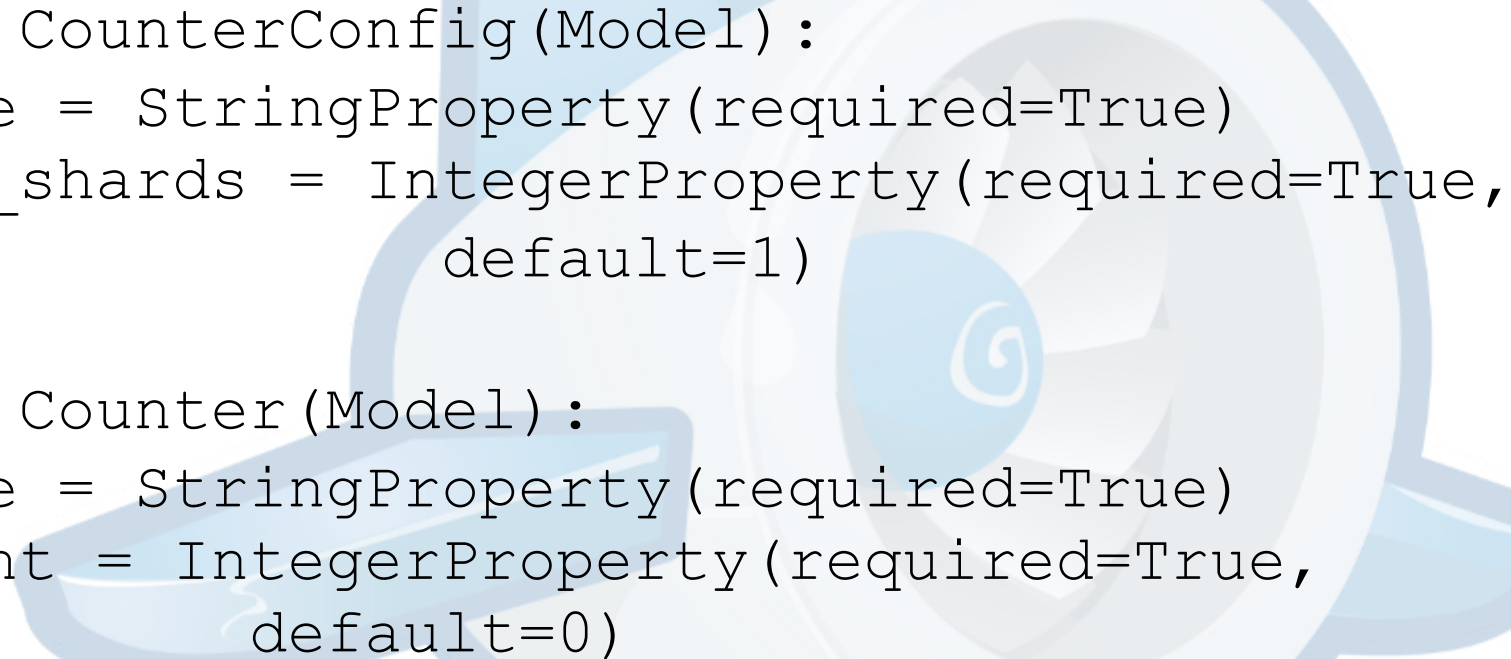
- ランダムでひとつ Entity を選びそれをトランザクショナルにアップデートする
- 読み込み時は全てを足して処理
- contention は $1/N$ に削減
- 分散させる要素数を変えるのはとても簡単



カウンター: モデル

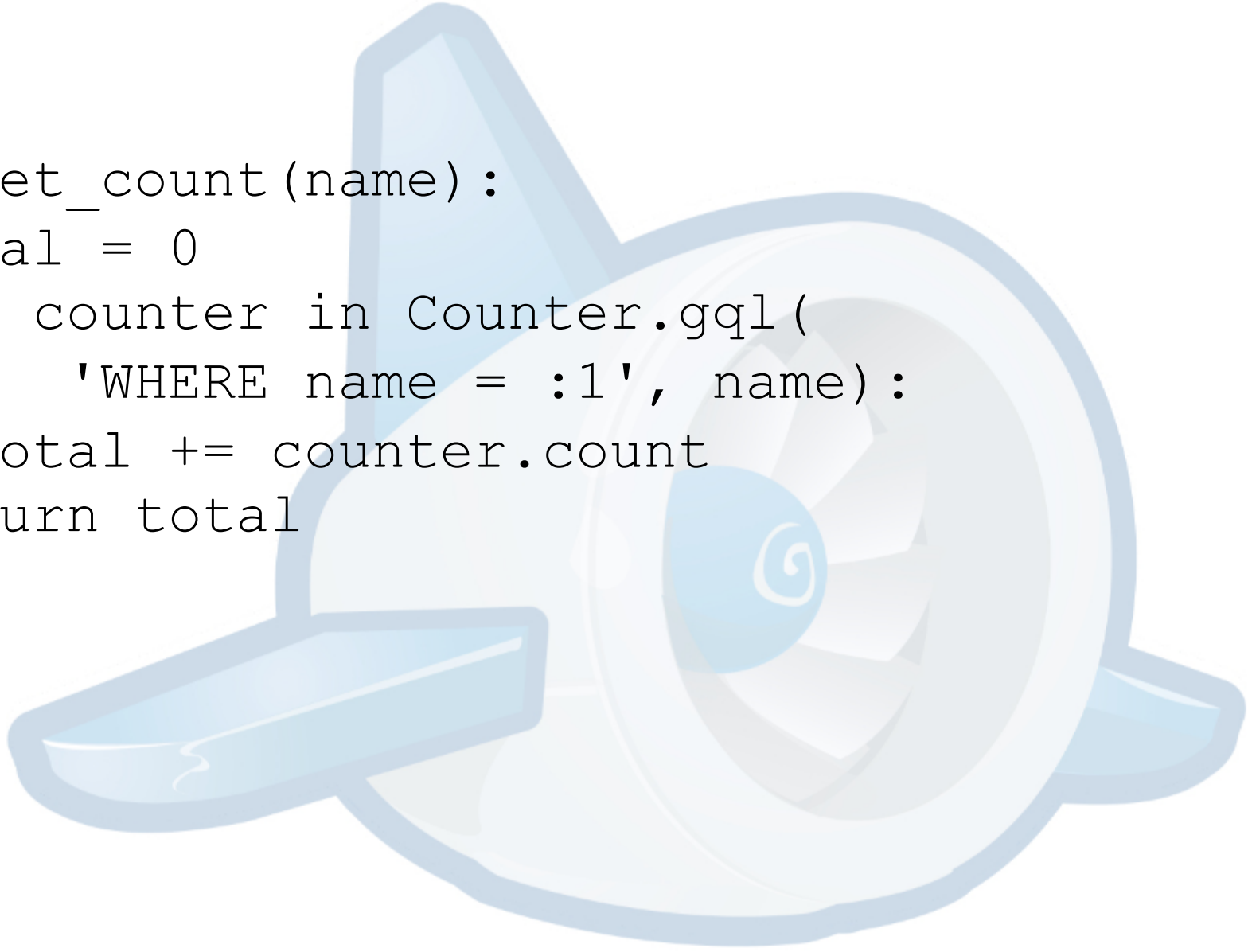
モデル

```
class CounterConfig(Model):  
    name = StringProperty(required=True)  
    num_shards = IntegerProperty(required=True,  
                                  default=1)  
  
class Counter(Model):  
    name = StringProperty(required=True)  
    count = IntegerProperty(required=True,  
                             default=0)
```



カウンター: カウンター値を得る

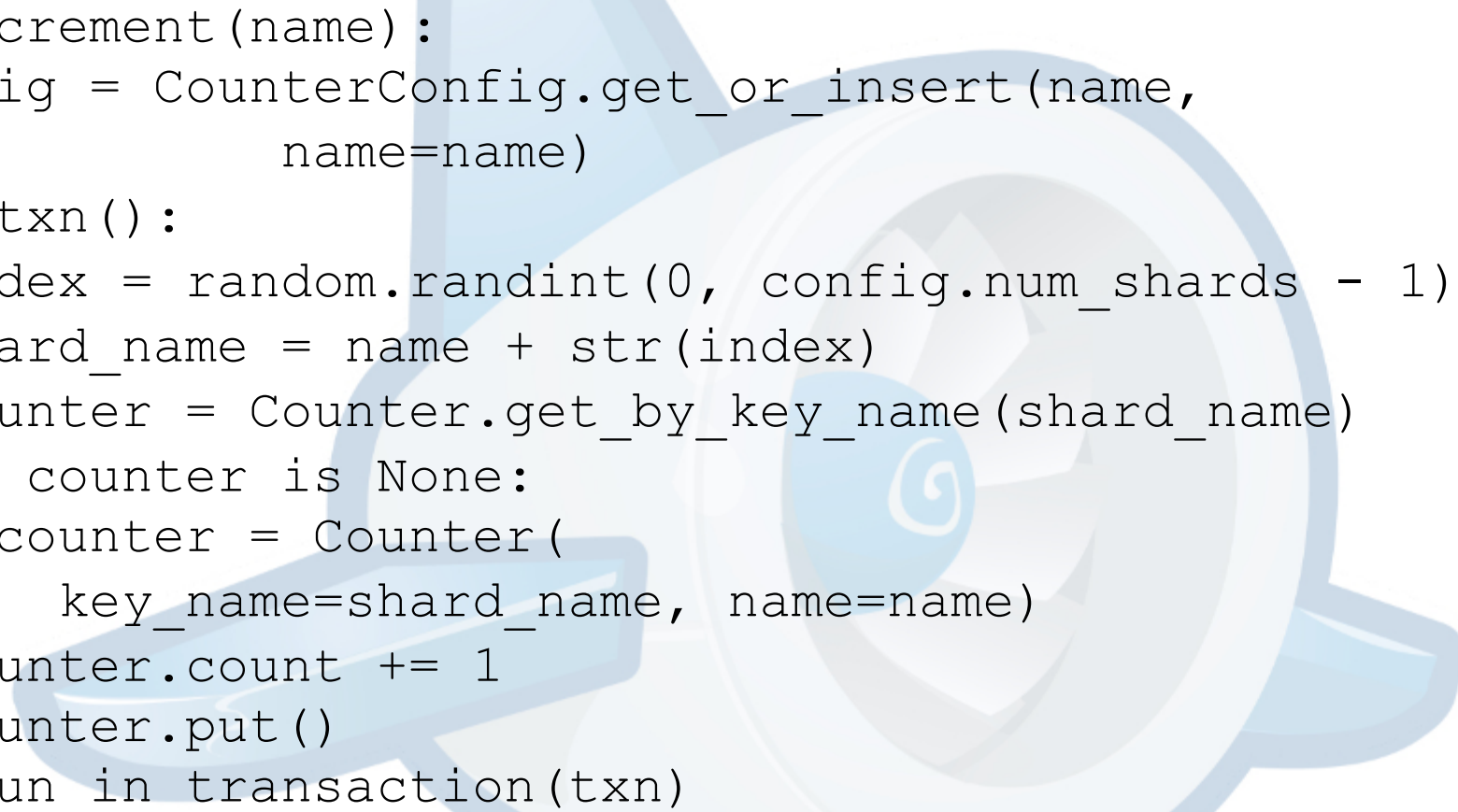
```
def get_count(name):  
    total = 0  
    for counter in Counter.gql(  
        'WHERE name = :1', name):  
        total += counter.count  
    return total
```



カウンター: インクリメント

```
def increment(name):
    config = CounterConfig.get_or_insert(name,
                                         name=name)

    def txn():
        index = random.randint(0, config.num_shards - 1)
        shard_name = name + str(index)
        counter = Counter.get_by_key_name(shard_name)
        if counter is None:
            counter = Counter(
                key_name=shard_name, name=name)
        counter.count += 1
        counter.put()
    db.run_in_transaction(txn)
```



カウンター: キャッシュを読む

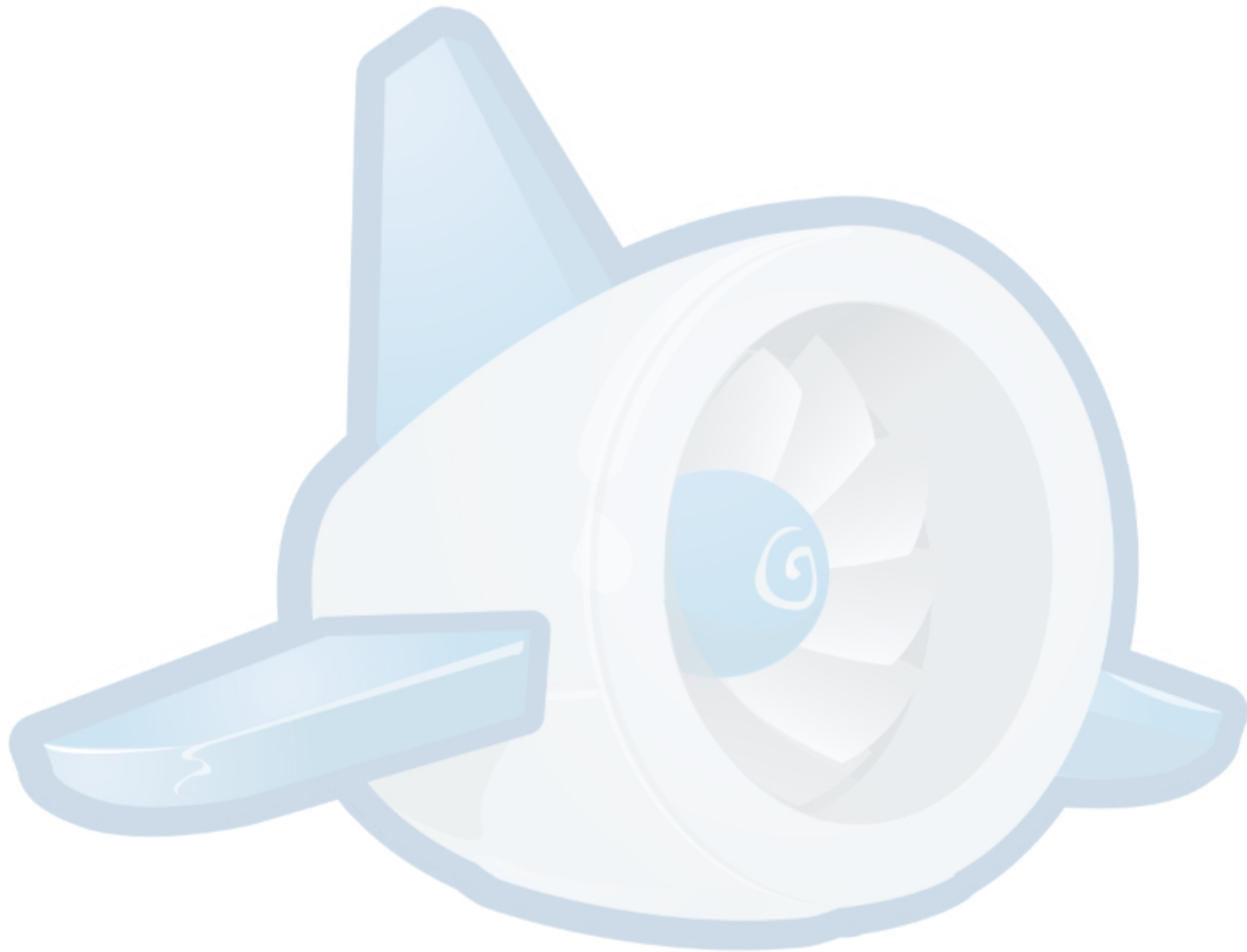
```
def get_count(name):  
    total = memcache.get(name)  
    if total is None:  
        total = 0  
        for counter in Counter.gql(  
            'WHERE name = :1', name):  
            total += counter.count  
        memcache.add(name, str(total), 60)  
    return total
```

カウンター: キャッシュに書き込み

```
def increment(name):
    config = CounterConfig.get_or_insert(name,
        name=name)
    def txn():
        index = random.randint(0, config.num_shards - 1)
        shard_name = name + str(index)
        counter = Counter.get_by_key_name(shard_name)
        if counter is None:
            counter = Counter(key_name=shard_name,
                name=name)

        counter.count += 1
        counter.put()
    db.run_in_transaction(txn)
    memcache.incr(name)
```

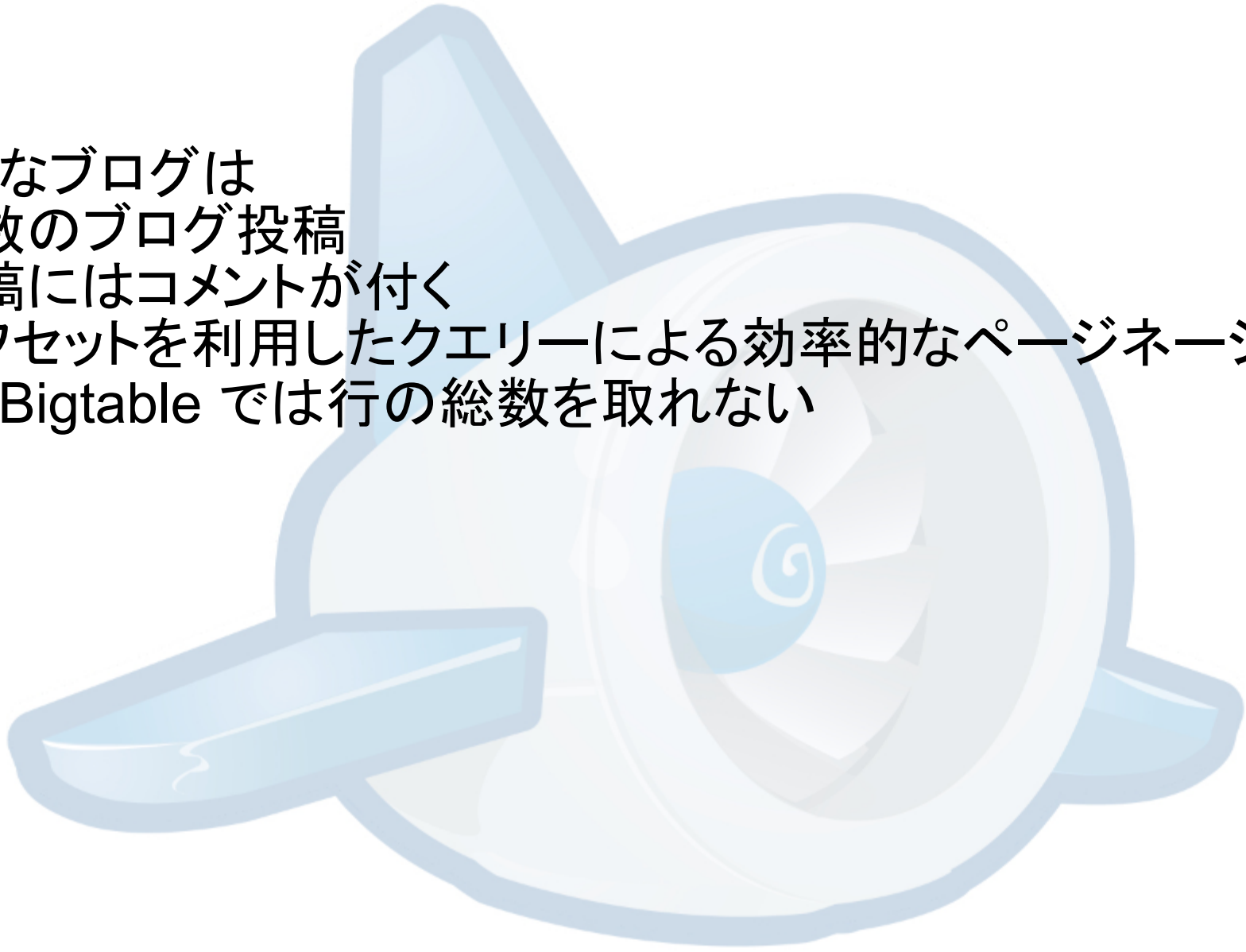
ブログ作成



ブログ作成

標準的なブログは

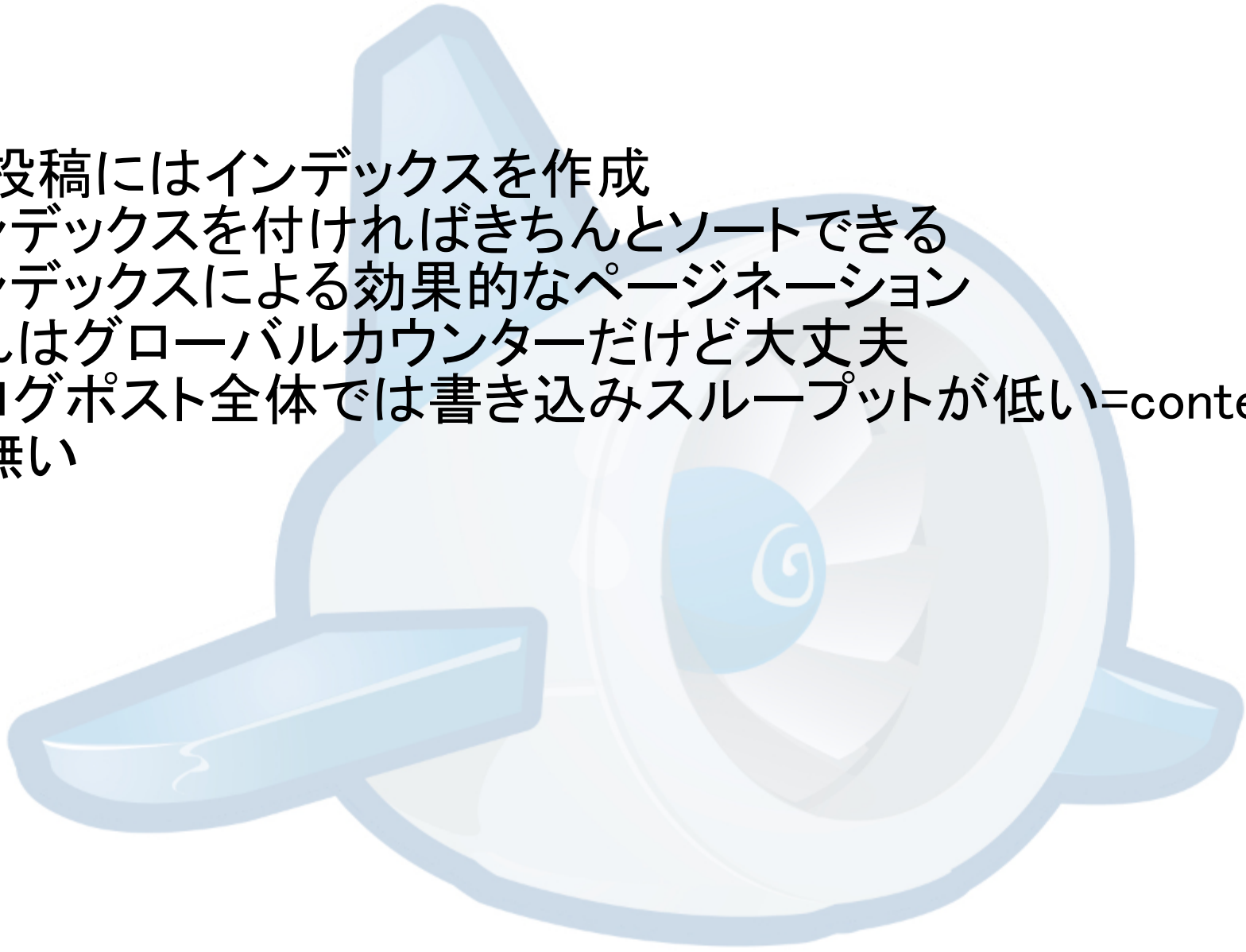
- 複数のブログ投稿
- 投稿にはコメントが付く
- オフセットを利用したクエリーによる効率的なページネーション
 - Bigtable では行の総数を取れない



ブログ作成: ブログ投稿

ブログ投稿にはインデックスを作成

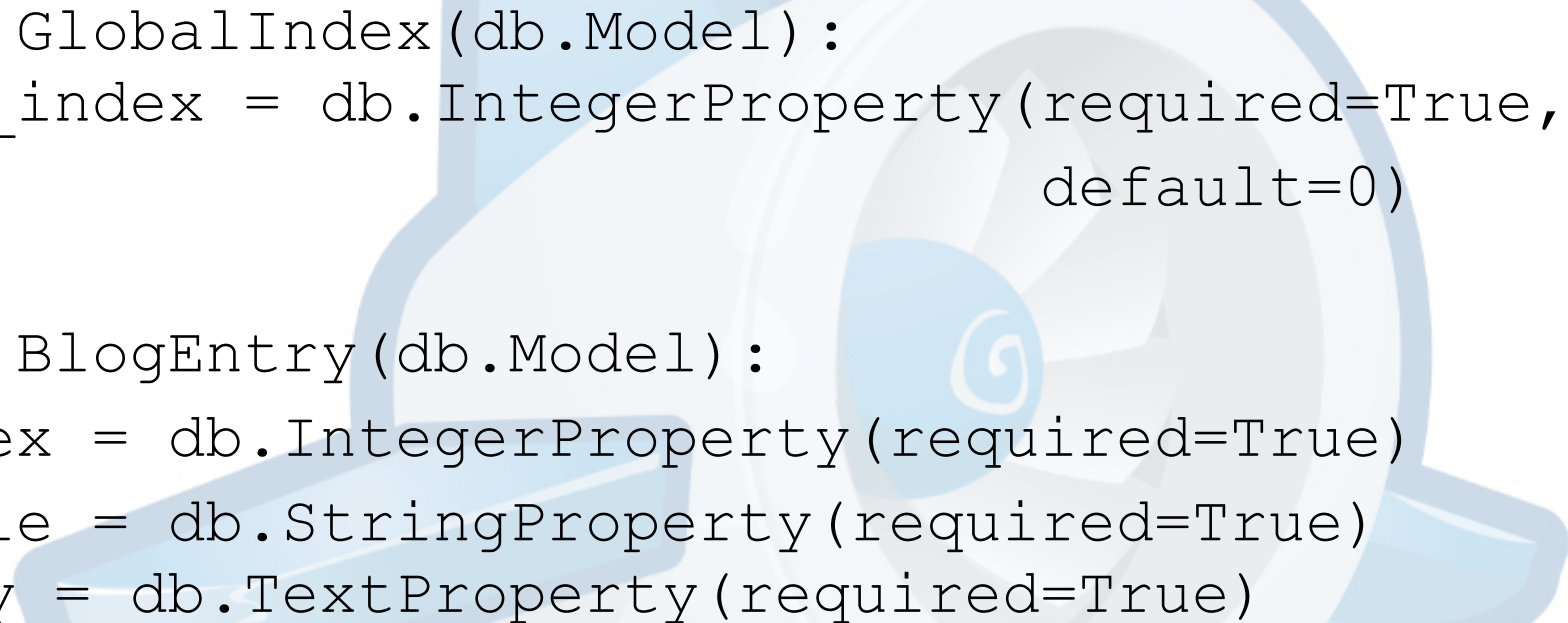
- インデックスを付ければきちんとソートできる
- インデックスによる効果的なページネーション
- これはグローバルカウンターだけど大丈夫
- ブログポスト全体では書き込みスループットが低い=contentionは無い



ブログ作成: モデル

Models

```
class GlobalIndex(db.Model):  
    max_index = db.IntegerProperty(required=True,  
                                    default=0)  
  
class BlogEntry(db.Model):  
    index = db.IntegerProperty(required=True)  
    title = db.StringProperty(required=True)  
    body = db.TextProperty(required=True)
```

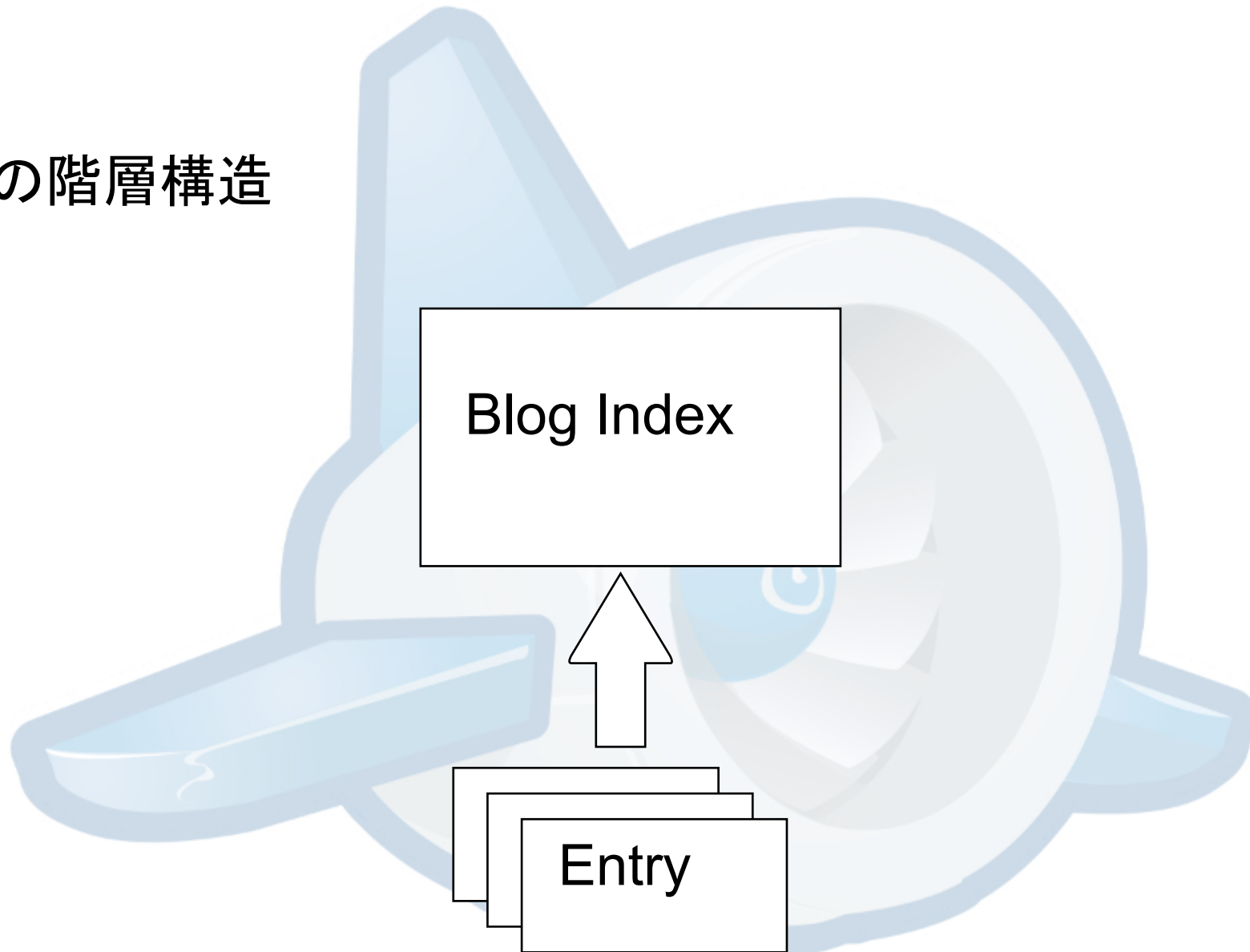


ブログ作成: ブログ投稿処理

```
def post_entry(blogname, title, body):
    def txn():
        blog_index = BlogIndex.get_by_key_name(blogname)
        if blog_index is None:
            blog_index = BlogIndex(key_name=blogname)
        new_index = blog_index.max_index
        blog_index.max_index += 1
        blog_index.put()
        new_entry = BlogEntry(
            key_name=blogname + str(new_index),
            parent=blog_index, index=new_index,
            title=title, body=body)
        new_entry.put()
    db.run_in_transaction(txn)
```

ブログ作成: ブログ投稿処理

Entity の階層構造



ブログ作成: ブログ投稿取得

```
def get_entry(blogname, index):  
    entry = BlogEntry.get_by_key_name(  
        parent=Key.from_path('BlogIndex', blogname),  
        blogname + str(index))  
    return entry
```

これだけ。むちゃ早い



ブログ作成: ページネーション

```
def get_entries(start_index):
    extra = None
    if start_index is None:
        entries = BlogEntry.gql(
            'ORDER BY index DESC').fetch(
                POSTS_PER_PAGE + 1)
    else:
        start_index = int(start_index)
        entries = BlogEntry.gql(
            'WHERE index <= :1 ORDER BY index DESC',
            start_index).fetch(POSTS_PER_PAGE + 1)
    if len(entries) > POSTS_PER_PAGE:
        extra = entries[-1]
        entries = entries[:POSTS_PER_PAGE]
    return entries, extra
```

ブログ作成: コメント

- 書き込みのスループットが高い
 - 共有インデックスは使えない
- 投稿の日時でソートしたい
- 日時はユニークでは無いので、ページネーションには使えない

2008-05-26	22:11:04.1000	Before
2008-05-26	22:11:04.1234	My post
2008-05-26	22:11:04.1234	This is another post
2008-05-26	22:11:04.1234	And one more post
2008-05-26	22:11:04.1234	The last post
2008-05-26	22:11:04.2000	After

ブログ作成: コメント

- 書き込みのスループットが高い
 - 共有インデックスは使えない
- 投稿の日時でソートしたい
- 日時はユニークでは無いので、ページネーションには使えない

2008-05-26 22:11:04.1000 Before

2008-05-26 22:11:04.1234 My post

2008-05-26 22:11:04.1234 This is another post

2008-05-26 22:11:04.1234 And one more post

2008-05-26 22:11:04.1234 The last post

2008-05-26 22:11:04.2000 After

ブログ作成: 合成プロパティ

独自の合成プロパティを作る

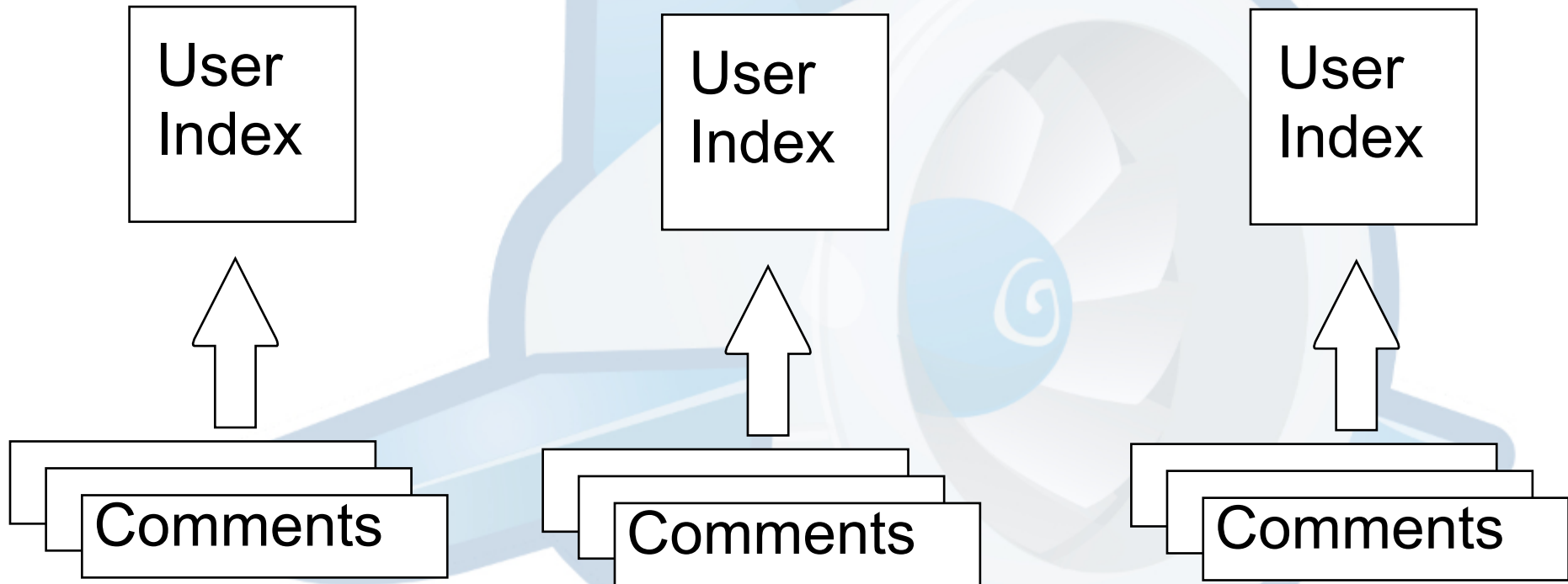
- "post time | user ID | comment ID"
- ユーザ毎の comment ID には共有インデックスを使う
 - そのインデックスは別々の Entity group にする

ユニークな順番が確保できる

2008-05-26	22:11:04.1000 brett 3	Before
2008-05-26	22:11:04.1234 jon 3	My post
2008-05-26	22:11:04.1234 jon 4	This is another post
2008-05-26	22:11:04.1234 ryan 4	And one more post
2008-05-26	22:11:04.1234 ryan 5	The last post

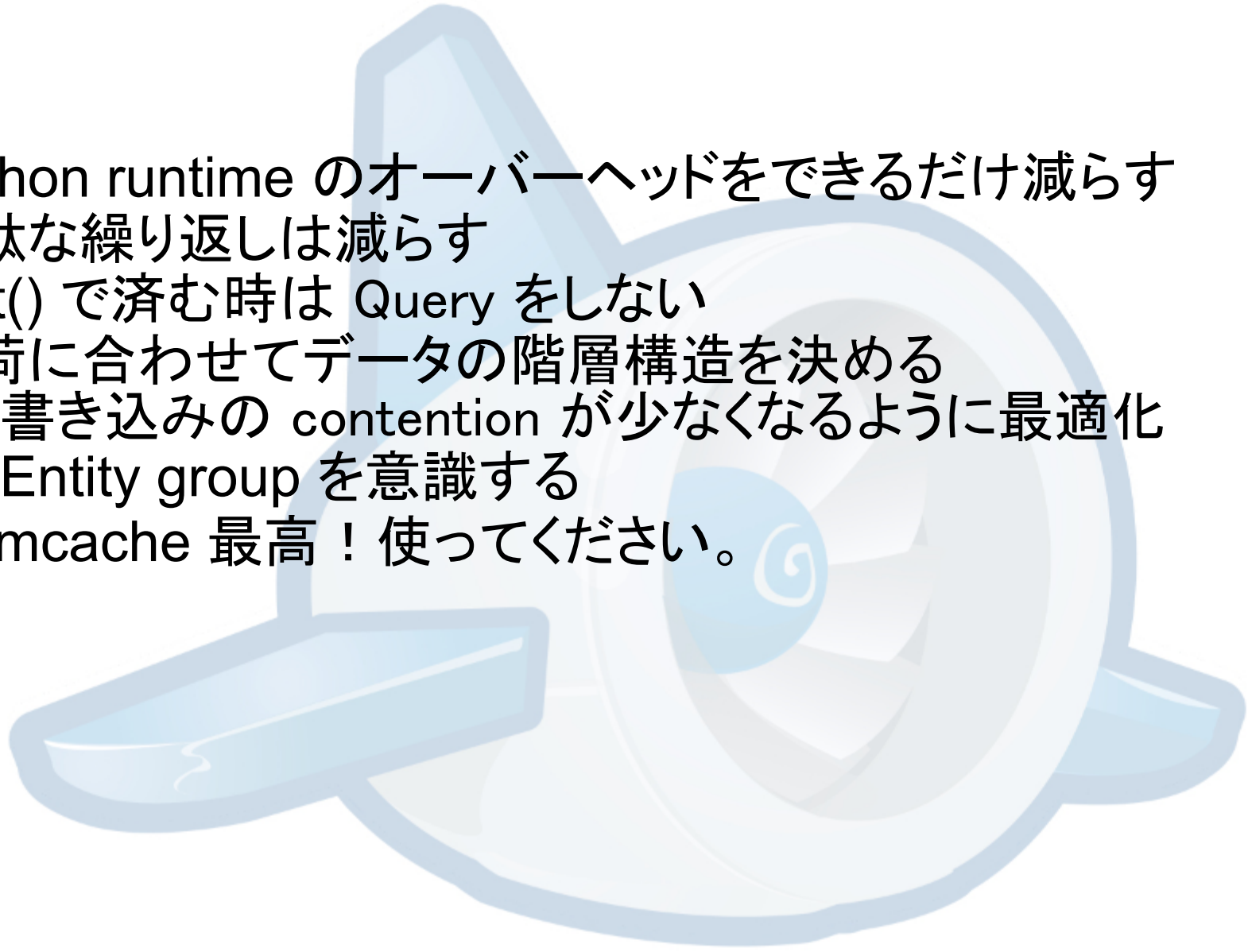
ブログ作成: 合成プロパティ2

並行した設計により高いスループットが得られる



覚えておく事

- Python runtime のオーバーヘッドをできるだけ減らす
- 無駄な繰り返しは減らす
- Get() で済む時は Query をしない
- 負荷に合わせてデータの階層構造を決める
 - 書き込みの contention が少なくなるように最適化
 - Entity group を意識する
- Memcache 最高！使ってください。



ありがとうございました

