

Next Gen Queries

Alfred R. Fuller May 20th, 2010 http://bit.ly/NextGenQueriesWave



Who?

3

- Software Engineer
- App Engine Datastore Team
 - Query Planner
 - Removed 1000 Entity Limit
 - Cursors/Prefetching
 - Python and Java SDK

Google[®] 10

Notes

- View live notes and ask questions about this session on Google Wave:
 - http://bit.ly/NextGenQueriesWave
- Good background in previous IO talks
 - Building Scalable, Complex Apps on App Engine
 - Google I/O 2009 Brett Slatkin
 - http://code.google.com/events/io/2009/sessions/ BuildingScalableComplexApps.html
 - Under the Covers of the Google App Engine Datastore
 - Google I/O 2008 Ryan Barrett
 - http://sites.google.com/site/io/under-the-covers-of-the-google-appengine-datastore



Outline

- Overview (of largest query changes since launch)
- Background
- Technology
 - Zigzag Merge Join
 - MultiQuery
- Examples
 - Just a few interesting cases (not necessarily practical)
- Corollaries



Current System

- Incredibly Scalable
- Schema-less
- Able to perform many query operations
 - Equality Filter (=, IN)
 - Inequality Filter (!=, <, <=, >=, >)
 - Sort
- Composite Indexes
 - Multiple filters (only 1 inequality) and sort orders
 - Logical AND operator
 - (not required for only equality filters and no sort)



Improvements – Query Capabilities

- Combine filters using arbitrary query logic!
 - AND, OR, NOT, sub expressions
 - (still single inequality filter)
- First class support for domain specific queries:
 - Geo-Query
 - Date Range Queries
 - (multiple inequality filters on numeric properties)



Improvements – Composite Indexes

- Reduced requirements
 - Solve exploding index problem
 - Fewer indexes server more queries
- Composite index selection = dark art
 - Space Vs Speed
 - Write Vs Read Latency/Cost



Scalability





Disclaimer

- Next Gen Queries != Current or Next Release
 - Rolling out these features in pieces
 - Infrastructure is there
 - Exposing it to users will take time
 - Syntax/API not finalized

• (some of the backend changes available today)



Example Model - Photo

- Owner: person who posted picture
- Tag: List of user assigned tags
- People: List of people in the photo (either user assigned or facial recognition)
- Location: Geo location of where photo was taken
- Date: Time/Date when the photo was taken









Background



Indexes

- Index data = Set of ordered values
- Index = Sorted collection of index data
- Built-in Property Index:

Kind	Name	Value	Key
 Composite Index: 			
Index	Compos	ite Value	Key



Query => Index Scan (Query Planner)

- Split index data into 2 pieces:
 - Prefix: held constant
 - Postfix: different for each result
- Equality filters => prefix
- Orders => postfix
- Inequality filters => range restrictions on the postfix



SELECT * FROM Photo WHERE tag = 'family'



- Prefix Constraints
 - Kind == Photo
 - Name == tag
 - Value == 'family'

- Postfix Ordering
 - key asc



SELECT * FROM Photo ORDER BY date DESC



- Prefix Constraints
 - Kind == Photo
 - Name == date

- Postfix Ordering
 - date desc
 - key asc



SELECT * FROM Photo WHERE tag = 'family' AND tag = 'outside' ORDER BY date DESC



- Index 1 on (tag, tag, date)
- Prefix Constraints:
 - tag == 'family'
 - tag == 'outside'

- Postfix Order
 - date desc
 - key asc



Zigzag Merge Join (Current)



Zigzag Merge Join

- Efficiently finds index postfixes common to multiple scans
- Produces intermediate false positive results
 - Skips past sections of non-matching entities
 - Tends to scale with size of requested results



SELECT * FROM Photo WHERE tag = 'family' AND tag = 'outside'



- Prefix Constraints
 - S1.Kind == Photo
 - S1.Name == tag
 - S1.Value == 'family'
 - S2.Kind == Photo
 - S2.Name == tag
 - S2.Value == 'outside'

- Postfix Constraints
 - s1.Key == s2.Key
- Postfix Ordering
 - key asc



MultiQuery (Current)



MultiQuery

- Combines multiple query result sets
 - Uses priority queue based on orders
 - (newly optimized to avoid priority queue when possible)
- Currently supports IN and !=
 - WHERE tag IN ['family', 'friends'] ORDER BY date DESC
 - WHERE tag == 'family' ORDER BY date DESC
 - WHERE tag == 'friends' ORDER BY date DESC
 - WHERE tag != 'beach' AND tag != 'coworkers'
 - WHERE tag < 'beach'
 - WHERE tag > 'beach' AND tag < 'coworkers'
 - WHERE tag > 'coworkers'

Means: Has a tag other than 'beach' and 'coworkers'



Next Gen Features



Zigzag Merge Join += Sort

- New features:
 - Merge on entire postfix (instead of just key)



SELECT * FROM Photo WHERE tag = 'family' AND tag = 'outside' ORDER BY date DESC



- Index 1 on (tag, date)
- Prefix Constraints:
 - S1.tag == 'family'
 - S2.tag == 'outside'

- Postfix Constraints:
 - S1.date == S2.date
 - S1.key == S2.key

- Postfix Order
 - date desc
 - key asc



Zigzag Merge Join += OR, NOT

- AND
 - S1.postfix == S2.postfix
- OR
 - No join constraint
 - Every postfix matches
 - Each postfix only seen once
- NOT
 - S1.postfix != S2.postfix for all S1.prefix and S2.prefix
 - Equivalent to set subtraction
 - NOT(a) == ALL a
 - Does not require an inequality filter!!
 - Intuitive results for multi-valued properties!!



Google[®] 10 [O

Not Example

- a AND NOT(b)
 - a = [2,3,5]
 - b = [1,3,4]
 - result = [2,5]
- Algorithm
 - a:2
 - b(>=2):3
 - 2 matches
 - a:3
 - 3 does not match
 - a:5
 - b(>=5):Null
 - 5 matches
 - a:Null
- Performance:
 - Only looked at 4 out of 6 keys to produce 2 results



((a AND b) OR c) AND NOT(d)





MultiQuery += OR

- WHERE tag IN ['family', 'friends'] OR person IN [...] ORDER BY date DESC
 - WHERE tag = 'family' ORDER BY date DESC
 - WHERE tag = 'friends' ORDER BY date DESC
 - WHERE person = '…' ORDER BY date DESC
 - WHERE person = '...' ORDER BY date DESC
 - ...
- No false positives but could be lots of dupes!
- Performance vs. Zigzag depends on data



MultiQuery += Geo, Date Range, ...

- QuerySplitter
 - Produces multiple parallel query components
 - Geo/Date Range: range scans along space filling curve
 - Accuracy Vs # of parallel components
- Entity Filter
 - Removes unwanted results
 - De-dupe always needed
 - Geo/Date Range: Fuzzy result pruning
 - Removes points outside of exact range



Examples (Current Vs Next)



SELECT * FROM Photo WHERE tag = 'family' AND tag = 'outside' ORDER BY date DESC

- Meaning: "Find recent photos of my family taken outside"
- Minimum composite index requirement:
 - Current Gen: Index on (tag, tag, date)
 - Repeated multi-valued property, "tag" (exploding index)
 - Next Gen: Index on (tag, date)



SELECT * FROM Photo WHERE tag IN ['family', 'friends'] AND tag = 'outside' AND tag != 'beach'

- Meaning: "Find all photos of my family and friends taken outside but not on the beach"
- Note: No sort order specified (will be ordered by tag ASC)
- Minimum composite index requirement:
 - Current Gen: Index on (tag, tag, tag)
 - Repeated multi-valued property, "tag" (exploding index)
 - Unintuitive results (tag != 'beach' will have no effect)
 - Next Gen: No index required
 - Uses built-in indexes



SELECT * FROM Photo WHERE (tag = 'family' OR tag = 'friends') AND tag = 'outside' AND NOT(tag = 'beach') ORDER BY date DESC

- Meaning: "Find recent photos of my family and friends taken outside but not on the beach"
- Note: Same as last query except ordered and using AND, OR, NOT and sub expressions
- Minimum composite index requirement:
 - Current Gen: Not possible because of order
 - Next Gen: Index on (tag, date)
 - Arbitrary number of 'tag =' filters supported by this single index



SELECT * FROM Photo WHERE (tag IN ['family', 'friends'] OR people IN [...]) AND tag = 'outside' AND NOT(tag = 'beach' OR location IN [{Coastal Regions}]) ORDER BY date DESC

- Meaning: "Find recent photos of my family and friends taken outside but not on the beach"
- Note
 - Not reliant on proper tagging
 - Using OR on different properties,
 - NOT is applied to a entire sub-expression
 - Requires clever use of geo encoding to order by date
- Minimum composite index requirement:
 - Current Gen: Not possible
 - Next Gen:
 - Index on (tag, date)
 - Index on (people, date)
 - Index on (geo(location), date)



N = size of largest sub-scan

- What to use when?
- Zigzag: produces intermediate false positive results
 - Tends to be O(R) worst case O(N)
- MultiQuery/Query: produces duplicate correct results
 - Guaranteed to be O(R) as # of sub-queries is constrained
- Actual performance depends on query and shape of the data
- Prefer MultiQuery/Query when possible
 - Not possible when indexes are missing or features are not supported
 - Can use both on a single query





Capabilities

- Composite Index Requirements
 - Zigzag: many, reusable, and simple
 - MultiQuery/Query: Requires index very specific index
- NOT
 - Zigzag: Any # of constrains on any # of properties + sort
 - MultiQuery: Any # of constraints on 1 property + first sort must be on that property

	Scalability	Index Requirements	AND	OR	NOT	Domain Specific Queries
Zigzag	O(R) – O(N)	Low	\checkmark	\checkmark	\checkmark	
MultiQuery/ Query	O(R)	High	Fixed # of filters	Restricted # of parallel queries	Single property	~



Planning/Optimizations

WHERE

```
tag IN ['family', 'friends']
```

AND tag = 'outside'

AND owner != 'al'

AND owner != 'nick'

NOT

```
owner = 'al'
```

```
OR owner = 'nick'
```



```
tag = 'friends'
```

OR

```
owner = 'al'
```

```
owner = 'nick'
```



³⁸ Google Confidential

Planning/Optimizations + Index (tag, tag, owner)

WHERE

```
tag IN ['family', 'friends']
```

AND tag = 'outside'

```
AND owner != 'al'
```

AND owner != 'nick'

tag = 'family' tag = 'friends'

owner < 'al'

```
'al' < owner < 'nick'
```

```
owner > 'nick'
```

WHERE tag = 'family' AND tag = 'outside' AND owner < 'al'

```
WHERE tag = 'family' AND tag = 'outside' AND owner > 'al' AND owner < 'nick'
```

```
WHERE tag = 'family' AND tag = 'outside' AND owner > 'nick''
```

```
WHERE tag = 'friends' AND tag = 'outside' AND owner < 'al'
```

```
WHERE tag = 'friends' AND tag = 'outside' AND owner > 'al' AND owner < 'nick'
```

WHERE tag = 'friends' AND tag = 'outside' AND owner > 'nick"



Planning/Optimizations + Index(tag, tag)



WHERE tag = 'family' AND tag = 'outside' WHERE tag = 'friends' AND tag = 'outside' WHERE owner = 'al' WHERE owner = 'nick'



Space Vs Time (Write Vs Read Latency/Cost)

- WHERE a = '...' AND b = '...' AND c = '...'
 - a = '…'
 - b = '…'
 - c = '…'
- WHERE a = '...' AND b = '...' AND d = '...'
 - a = '…'
 - b = '…'
 - d = '…'
- + Index(a, b, c)
- + Index(a, b, d)
- +Space, -Time, +Write, -Read



Already Available

- + Index(a, b)
- WHERE a = '...' AND b = '...' AND c = '...' - a = '...' AND b = '...' - c = '...'
- WHERE a = '...' AND b = '...' AND d = '...'
 - a = '...' AND b = '...'
 - d = '…'



Corollaries



SearchableModel becomes useful!

- Every query can now be solved using zigzag
 - Never a need to have more than one property in prefix (although can be good to improve performance)
- Almost never a reason to sort or restrict range on multivalued properties
- Can handle arbitrary number of equal filters without needing any extra indexes
- Thus no more exploding indexes!



Cursors

- Only store postfix
 - Much smaller
 - Can be used on any index with the same postfix
 - Equality filters can change completely
 - Can be used with multi-query
 - All queries in multi-query have the same postfix
 - Deduping issues
- Store raw values
 - Positions can be generated directly from an entity if there are no multi-valued properties in the postfix
 - A query with reversed orders can be used to scroll backwards
 - Requires trailing key descending order
 - Always requires extra indexes



Questions? http://bit.ly/NextGenQueriesWave

