Project 2020:
Develop a Graph Database using a NoSQL Cloud-Native Database as the datastore.

Author: Ross Payne

Date: April 2021

Table of Contents

Who should read this? 4
Software Stack 4
The Project Aims 4
Project Results 5
What is a Cloud-Native Database? 5
NoSQL vs SQL Databases 6
Why Dynamodb? 8
GoGraph Limitations 8
GoGraph's Code Dependencies 9
The Type System 10
Type Mappings 13
RDF Data 15
A Concurrent RDF Loader Design 17
Storage Abstractions 20
Block Types 21
Design Highlights 22
The Million Subscribers Problem 24
Table Key Design 25
The Data Model 27
Handling Nulls 31
Reverse Edge Design 31
More Details about the UID Overflow Design 33
Data Example: Person Node from Movie Graph 34
GoGraph Query Performance 37
Non-Concurrent Query Engine 37
Node Cache 37
A Word About The Data 37
Test 1: Edge Predicate Filter with Boolean expression 38

Test 2: Full Text Searching and Root Filter Expression 41
Test 3: Generate a Graph of Depth 5 44
Test 4: Has Function in Filter 48
Test 5: Has function in Root Filter and Edge Predicate 50
Test 6: AnyOfTerms function in Root Filter 52
Test 8: AllOfTerms function in Root Filter 54
Test 9: Equality Expression in Root Query 56
Test 10: AnyOfTerms Function used in Edge Predicate Filter 58
Test 11: Propagating Child Scalar Data to Parent 60
Test 12: Propagate scalar data to grandparent for 1:1 relationship 63
Test 13: Disabled child-to-grandparent propagation for 1:1 relationship. 70
Appendix A : CSP Programming in Go 75
Appendix B - How AWS Charges for Dynamodb? 78
Appendix C - Scalability and shared resources 79

Who should read this?

Those interested in designing a NoSQL database application, the performance capabilities of *Dynamodb* and CSP programming in Go.

Software Stack

Quick review of the technology stack that will be employed.

Component	Provider	Description
Public Cloud	*AWS*	
Databases	*Dynamodb*	Cloud-native datastore for graph data and type definitions. Full query support except full text search capability.
	ElasticSearch	Provides full text searching across all text attributes. Returns node UUID (universal unique identifier) which is passed into *Dynamodb* query. While ElasticSearch exists as a cloud-service from a number of cloud vendors including Elastic, AWS, Azure, for the purpose of this project a single server implementation was used sourced from a Bitnami AMI (Amazon Machine Image) running on a small EC2 instance.
Language	*Go*	A wonderfully pragmatic and productive C-styled language designed by a small team of software luminaries, some of which were responsible for C and Unix, and supported by none other than Google. With a CV like that why would you not use it. It is also one of a few languages that supports *Communicating Sequential Processors* (CSP) natively, which in the age of multi-core CPUs provides the developer with an easy way to scale the application.
Testing	*go test command*	All testing used Go's command line tool *go test*
Source Control	*Git*	

The Project Aims

This aims of the project are detailed in the table below:

Aim	Outcome
As an exercise in improving my Go programming, following the idea that rhe more you program in Go makes the better Go programmer you become.	*GoGraph*, a rudimentary graph database with GraphQL* like query language, node cache and concurrent RDF loader.
Use a NoSQL database to design a Graph database	Create a Graph Data model in Dynamodb
Make a serious attempt at designing a graph database using AWS's *Dynamodb* NoSQL database as the data store. All aspects of the data design should should take advantage of the distributed nature of cloud databases and their advantages in scaling.	A highly scalable graph database designed enabling run queries on nodes with millions of edges to be run in a handful of seconds.
Incorporate a Type system	Graph schemas stored in the database
Incorporate full text search capability	Simultaneously load RDF data into Dynamodb and ElasticSearch. Define ElasticSearch query functions.
Incorporate a non-blocking data cache	API to Fetch Nodes from cache.Locking AP ???? Disable for testing as we are testing Dynamodb fetch performance not the cache performance.
Create a functional test suite of GraphQL queries.	Publish results in this document
Quantify query performance	Run some demanding graph queries

* Syntax and some functions based on DGraph's GraphQL

Project Results

All the aims listed in the Project Aims have been achieved and are discussed at length in this document. However to highlight the most significant design aspects in *GoGraph*:

* **Cluster node data around the node's UUID** (Universal Unique Identifier): which becomes the table's partition key. However very large items, like edge data (aka UID edges) with millions of child nodes, will be offloaded to overflow blocks which will spread over multiple partitions for scalability purposes

* **Partition a node's data based on sort key:** a sort key design has been implemented, that enables queries to target individual parts of a node's data e.g. all or a subset of the scalar data, all edges or subset of edges, only reverse edge data etc to minimise query response times and database read costs (actual money).

* **Duplicate scalar data** (aka scalar propagation) from child to parent node and additionally, child to grandparent node, in the case of 1:1 edge cardinality between child and parent node. Scalar propagation improves data clustering and enables columnar compression.

* **Incorporating two data stores:** *Dynamodb* for general graph queries and *ElasticSearch* for full text queries. *GoGraph* seamlessly integrates the two.

* **Overflow UID blocks:** enables highly connected data, potentially with millions of edges per node, to scale across *Dynamodb* physical partitions thereby enabling *GoGraph* to parallelise queries with minimal IO contenion.

Onto the performance of GoGraph and the effectiveness of *Dynamodb*. Ideally it would be useful to directly compare GoGraph's performance to other graph databases by loading in the same data and execute the same queries in whatever language the alternate graph database used. However this would be a project in itself so I will simply rely on the performance metrics from each test using the Movie data to provide some level of judgement on performance. On this basis the performance is more than adequate I believe.

Note: there were no external or application level data caches employed in any of these tests. GoGraph disabled its internal node cache for all these tests. Queries therefore must source all node data from the databases, Dynamodb and/ or ElasticSearch.

Examples:

* the **Dynamodb Query API** for a single node takes between **3.5ms and 4.5ms** for 90% of the queries. Because of scalar propagation a single node query can return scalar data from all child nodes across each edge predicate, so a 3.5ms query can return not only the node data but also all its child node's scalar data which will benefit any filter condition as the data is already present in memory.**?????**

* Test 11 demonstrates the power of duplicating data for edges with a 1:Many cardinality. It shows 90 nodes effectively returned from 6 Dynamodb read calls with an **average response of 55 us (microseconds) per node**

* Test 12 demonstrates the power of duplicating data from child to grandparent for edges with 1:1 cardinality between child and parent. It shows **1166 nodes can be effectively queried in 122 ms for an average of 104 us (microseconds) per node**.

* **ElasticSearch** queries took between **7ms and 8ms**.

What is a Cloud-Native Database?

At a high level a cloud-native NoSQL database must be **elastic, highly resilient and fully managed.** To achieve this it must implement the following:

Feature	Comment
Be Distributed	The database must run across multiple servers. An essential attribute for scaling and resilience
Scale horizontally on demand	The database must have the ability to automatically add (and remove) compute and storage servers in response to database load and distribute both the data (e.g. table partitions) and the load over the additional resources.*
Makes use of elastic infrastructure	The underlying infrastructure must not pose any limits on the resource requirements of the database and its ability to scale on demand.
Replicates Data	For availability and resilience purposes the database must replicate across storage servers and availability zones.
Nothing to provision	A cloud native NoSQL database should always be on, available and accessible via an API with no requirement to provision resources on behalf of the user *
Charge by usage	Charge by usage for any services used. Services may include backup and restore, streaming, external caching, auto scaling. Storage costs are charged by amount by period.
Self-Healing	The database cluster should detect failed servers and replace them automatically. The database should also be able to detect table hot spots and automatically implement strategies to reduce its impact.
Detailed Monitoring	The underlying database infrastructure should be monitored with sufficient detail to drive any scaling design decisions or analysis of a failed operation. ???
Automatic Backups	The database should be backed up automatically without human intervention
Advanced Security	Data should be encrypted both at rest and in transit. There should be a thorough identity and access management system in place to define and control privileges.
Consistently achieve single-digit millisecond response at any scale	Ultimately most of the above is aimed at providing a foundation for achieving consistently fast query response.
Automatic patching and upgrades	The database software should apply patches automatically without any outages.

* This is in contrast to some cloud-native SQL databases that require the user to first provision a cluster of compute nodes. The storage nodes are a separate service that scale independently and automatically from the compute.

These features are not necessarily the sole domain of NoSQL databases as some of the latest SQL databases, e.g. Google's Spanner and AWS Aurora, exhibit many, if not all, of the attributes of a cloud-native database except the requirement the no-provision requirement.

Cloud-native NoSQL databases are available on all the big three public cloud providers, *Dynamodb* from AWS, *Cosmos DB* on Azure and *Google Cloud Datastore* on GCP (Google Cloud Platform). There are also a number of open-source cloud-native offerings, such as Datatax *Casandra and Mongodb* DB *Atlas*.

NoSQL vs SQL Databases

NoSQL databases are perceived to be cheaper, faster and scale significantly better than their fancier feature rich SQL cousins.

Some of the justification for this view is supported by AWS's NoSQL offering, *Dynamodb who's raison d'etre* was the failure of traditional SQL databases, notably Oracle, to scale to the extreme loads required of amazon.com shopping site and in particular the annual Black Friday sale event. SQL databases would fail under such extreme requests rates ultimately leading to well publised outages. Amazon created Dynamodb to replace the high frequency SQL database queries with their NoSQL equivalents.

Why is it that NoSQL can scale better than a SQL database? Is the problem entirely because of SQL or is there more to it?

It's pretty easy to run an experiment that will show the limits of an Oracle database (or any other "traditional" SQL database for that matter) limits……

However it is not all upside for NoSQL databases as they are also perceived to have a narrower range of use cases compared to their SQL counterparts which offer a more familiar implementation of the relational model, ACID transaction, higher levels of data consistency.

Over the last twenty years, small commodity based servers where increasing in power sufficient to take on some enterprise workloads while universities where frantically publishing research papers describing how to cluster thousands of commodity servers into large, powerful, highly available and resilient systems to build vast distributed storage systems, HPC and large batch processing systems. In response commercial and open source software designers either re-architected or architected their products from the ground up to be distributed while at the same time data centres started filling up with commodity servers attached to fast low latency networks and SSDs. This was a match made in heaven for the NoSQL database designer.

Software designers also knew each database feature, while useful, came at a measurable cost to scalability[1]. Why? Each feature adds to the number of "shared resources" (such as caches and locks) that need to be managed via a mutex and under extreme load mutex waits can start and dominant all other waits leading to potential scalability issues[2]. This was a clue to NoSQL database designers to be judicious with features to the point of eliminating all but the essentials.

So, the "No" in NoSQL not only means no SQL, it can also mean:

* no data caching (yes, every request is a physical IO),
* no schemas,
* no database verification of data relationships,
* no sequencer generators,
* no database locking mechanisms

NoSQL relatively tiny feature means less code and therefore a large reduction in the number of speed sapping "shared resources" that needs to be managed by the database. For this reason NoSQL is synonymous with a bare bone database designed to be fast because of what it doesn't do.

In the case of Dynamodb, and I presume other NoSQL's databases, the faster response can also be attributed to the use of the ***index organised table*** (IOT) structure for tables. IOTs have the benefit that all key and non-key data is clustered together enabling all related data to be accessed in one read request. SQL databases on the other hand use a heap structure for table data with separate index structures for each lookup key. As a result fetching from a SQL table requires two read operations unless the index also includes all the required non-key data. Consequently what is performed in one physical read operation of a

---

[1] See Appendix C for a explanation and example

[2] See Appendix C for further explanation of why a mutex is required and mutex waits

NoSQL database typically requires two physical reads in a SQL database[3]. Note: some SQL databases do offer an IOT as an alternative to the heap table, and in the authors opinion, are not used enough.

Finally, the ***schemaless*** nature of a NoSQL database means all related data, with different attributes, can be clustered together under the same partition key value in a single table. The immediate advantage is all related non-key data can be loaded into application memory in a single database read request[4]. This compares to the "normalised" SQL database design which allocates a table for each data type, requiring one more more IO reads per table to orchestrate the data into program memory. Some SQL database's offer a "clustered table" as a design option which attempts to emulate NoSQL's schemaless design advantage by physically pre-joining table data at the storage block level. However, there are so many constraints on its implementation clustered tables are rarely used in my experience, at least with Oracle.

The clustering benefits of both *IOTs* and *schemaless* design combined with NoSQL shorter code paths and lack of mutex's all combine to put it ahead of SQL databases in scalability and absolute speed of response.

Why Dynamodb?

There is a daunting array of proprietary and open-source database technologies available today with an equally daunting set of capabilities. *Dynamodb* however, has the considerable advantage that it has been battle tested in what is possibly the worlds biggest shopping site amazon.com, and as a result has received a laser like focus on continual improvements to scalability, raw performance and operational resilience like no other NoSQL database[5]. The focus on continual improvement ensures *Dynamodb* can cope with the next Black Friday sale.

Because *Dynamodb's* design is heavily biased towards scalability and raw performance it has had to judiciously sacrifice any feature that would get in the way of that goal[6]. Data consistency has however not been sacrificed and strong consistency can be specified for any query when required, otherwise eventual consistency is the default.

So as a cloud-native datastore for a Graph database, *Dynamodb* must be at or very near the top for raw performance. If GoGraph does not perform I cannot blame the datastore.

Finally, I have used *Dynamodb* in previous projects and can attest to its simplicity in development, it's always on availability and its ability to consistently respond to queries with single digit millisecond response times.

GoGraph Limitations

As this is a personal exercise and not an open-sourced project, I have restricted the features in GoGraph and GraphQL[7] to the bare minimum to provide a reasonable test for the effectiveness of *Dynamodb* as a datastore for Graph queries.

---

[3] Of course some of those reads may be from the data cache that SQL databases use. Depending on the type of application (OLTP for example) the cache may be beneficial for other applications (batch processing, analytics) it may add little to no benefit and may in fact slow the overall response because of the mutexes associated with the data cache.

[4] depending on the amount of data returned their maybe multiple Read Capacity Units consumed.

[5] For example, the "adaptive capacity" features is a recent resilience improvement

[6] While SQL is supported it is part of its code base, rather it is supported via a PartiQL client driver that simply issues standard Dynamodb API calls.

[7] The GraphQL language developed for GoGraph is based on DGraph's (DGraph.io) GraphQL language

GoGraph Feature	Supports	Restrictions
Type System	Node and Edge Facets Nested types Syntax for edge cardinality ie. one to one and one-to-many edges. Multiple graphs supported	
Concurrent RDF Loader	Configurable degree of concurrency	File size limited to available memory on server. No support for edge facets. No ACID compliance
Graphs	Multiple graphs Graphs consists of nodes and edges (node->child) Each node has a type Unlimited number of nodes supported All scalar attributes are indexed Each node has any number of scalar edges (attributes) Each node has any number of UID predicates (node-node edges) Each UID predicate connects unlimited child nodes Each child node maintains reverse edges	No support for Edge Facets No support for Graph Algorithms
Query Language	GraphQL based on DGraphs (DGraph.io) query language Query applies to a graph nominated at parse time. Root Query with Filter Filter Edge (on child node scalar data) Root & Filter functions: has,gt,lt,eq,le,ge Boolean Filter operators: AND, OR Full Text Search: via ElasticSearch	No support for querying Facets No support for Mutations No support for Transactions No support for NOT Boolean filter operator

---

GoGraph's Code Dependencies

The projects **go.mod** file, (https://github.com/rosshpayne/GoGraph/go.mod) also shown below, lists GoGraph's dependencies. The only dependency outside of AWS and ElasticSearch is a package responsible for generating Version 4 UUIDs.

```
module github.com/DynamoGraph

go 1.13

require (
	github.com/aws/aws-sdk-go v1.34.9
	github.com/elastic/go-elasticsearch/v7 v7.9.0
	github.com/satori/go.uuid v1.2.1-0.20180404165556-75cca531ea76
)
```

The Type System

Github: https://github.com/rosshpayne/GoGraph/types

Each node in a graph is an instance of a single type defined in the type system. The assignment of a node to a type is defined in the graph's RDF file, which is describe in detail in the next section. While a node can only have the attributes of its assigned type there is a work-around that enables a node to be effectively a composite of multiple types - more on that later.

A type can only belong to a single graph. The definition of each type is composed of ***Scalar attributes*** and/or ***Nested-type attributes***. A *Scalar attribute* is either a *number*, *string* or *binary* type or one of their *set* counterparts. The type of a *Nested-type attribute* matches an existing type in the type system. In the case of nested-types, if the type name is enclosed in [ ] it represents a 1:Many cardinality, if it is not enclosed in [ ], it represents a 1:1 cardinality, meaning the associated predicate will only ever have one edge (See RDF section below).

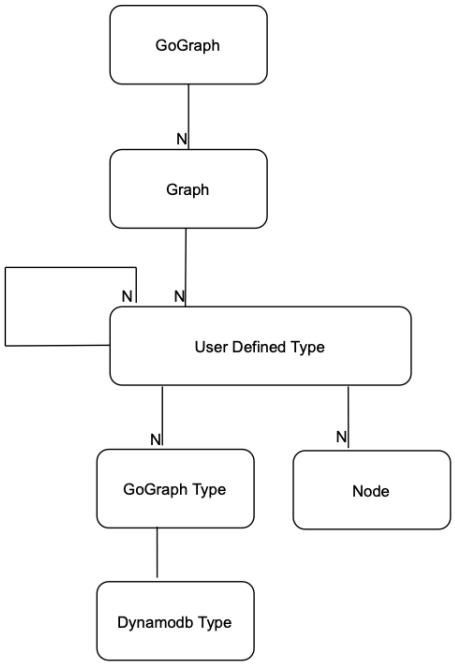
To expedite development there is no DDL language for types, rather each type is defined as a JSON entry in file and loaded into a single Dynamodb table using the AWS CLI:

```
aws dynamodb batch-write-item \
  --request-items file://<typeFile>.json \
  --return-consumed-capacity TOTAL
```

For an example of a JSON type definition file, see https://github.com/rosshpayne/GoGraph/json/Types.Movie.json . This file defines the type that are used in the Movie graph. The associated load script: https://github.com/rosshpayne/GoGraph/json/batchWrite.sh

The *Type* table is scanned into multiple caches (Go maps) as GoGraph's initialises during startup. Thereafter, all type data is sourced from the caches.

GoGraph supports multiple independent **graphs**, as depicted in the diagram below. Each graph in turn is defined with one or more **user-defined types**. Types cannot be shared between graphs (although that sounds like a good idea), consequently a Person type in Graph A can be different to a Person type in Graph B.



Each instance of a user-defined *type* is represented by a **node**, which like a *type,* can belong to only one graph. As mentioned, a node can only be defined as a single *type*, however, if the node type contains one ore mote nested types with a cardinality of 1:1, it is possible to emulate a node made up of a composite of types

RDF to Type Mapping

Graph data is defined in a file using the RDF format, **subject-predicate-object (s-p-o),** and loaded into *Dynamodb* using *GoGraph's Loader* program. Each node is represented by the set of RDF triples containing the same subject value. The RDF triple that defines the type for a node is identified by the predicate value "*__type*" where the object value stipulates the type name, as defined in GoGraph's type system for the particular graph. All other RDF triples, that match on subject, defines either one of the attributes of the node type, referred to as the ***scalar predicates*** or defines an edge (node to child) when the predicate matches a nested type as defined in the GoGraph type specification, in which case it is referred to as a ***Edge Predicate,*** as it will eventually be defined using a UUID value during the loader process.

Schemas

A graph's schema is represented by all the types that have been assigned to the graph.

The *Type* Data Model

All type related data, from the name of a graph, to its associated types, is stored in a *Dynamodb table called GraphSS.*

GraphSS has a composite primary key, composed of a *partition key* (attribute name: *PKey,* type *string)* and a *Sort Key* (attribute name: *SortK,* type *string)*. The *PKey* value is used to define the various categories of type data, as listed in the first column of the table below, while the *SortK* and other non-key attributes are used to define instances of the category data. For example, the first entry in the table below shows that the PKey value of "#Graph" is used to define the name of a graph, both its short and long version. *SortK* contains the short name while the non-key attribute*, Name,* contains the long name. The entry in red shows an instance of this data, "m" is the short name for a graph called "Movies".

The names of the types that belong to a graph are associated with the PKey value of "T" prefixed with the graph's short name. Again, each type name has a long and short version. There is an example below of type Person which is given a short name of P.

The attributes associated with each type are identified with a PKey value of the *type name* prefixed with the *graph's short name.* There are examples of all the attributes associated with the Person type for graph Movies.

Category	**GraphSS *PKey value*** ***Example***	**GraphSS *SortK value*** ***Example***	**Associated non-key Attribute(s)** ***Example***
Graph Name	**#Graph**	<graph short name>	Name (Graph Name)
	#Graph	*m*	*Movies*
Type Name *Must match the RDF object name for a predicate value of "__type".*	**#<graph short name>.T**	<type short name>	Name **(type name)**
	#m.T	*P*	*Person*
Type Attributes *Must match predicate names in RDF's "subject-predicate-object"*	**<graph short name>.<type name>**	<attribute name>	
	m.Person ***m.Person*** ***m.Person***	*name* *director.film* *performance.actor*	See table below…

The non-key attributes of table GoGraphSS that used to define each type attribute are listed in the table below.

GoGraphSS Column	**Type Attribute**	**Dynamodb Data type**	**Nullable**	**Values…**
F	Facet	String Set	Yes	*Not implemented.*
C	Short Name	String	No	Keep to single char if possible, as GoGraph will embed the attributes short name in all sort key values of the graph instance data (see Sort Key format)

GoGraphSS Column	Type Attribute	Dynamodb Data type	Nullable	Values…
N	Nullable	Boolean	No	Used by GoGraph to determine if attribute data must be present. Will cause the Load program to error otherwise.
Ty	Associated DynameoDB data type	String	No	***Scalar*** *type:* "S" String *length determined by database* "I" Integer *int64* "F" Float *float64* "B" Binary *length determined by database* "Bl" Boolean "DT" DateTime ***Set*** *type: an unordered collection* "SS" String Set "NS" Number Set "BlS" Boolean Set "BS" Binary Set ***List*** *type: an ordered collection* "LS" String List "LN" Number List "LBl" Boolean List "LB" Binary List ***UID*** *type: denotes a node-node edge* For cardinality 1:1 "Type ID" e.g. Person For cardinality 1:M "[Type ID]" e.g. [Person]
P	Partition ID	String	No	Used to partition Node data into blocks (see Storage Abstractions). Values **"A". . "F"** are reserved for **Scalar-attributes**. Default is "A". It is good practice to partition the data when there are tens or hundreds of attributes that are not always queried together or an attribute contains a lot more data than others and is rarely queried. Values **"G". . "K"** are reserved for **Edge-Predicates** ie. edge data**.** Default is "G". This allows the designer to group *Edge-Predicates* in queries and filter node data at the database rather than the application.
Pg	Propagate	Boolean	No	Should the attribute value be propagated to its parent. Default is True.
Ix	Index	String	Yes	Extend GoGraph's default index behaviour which is to automatically add scalar data to a GSI 'X' - for Set types only. Will add entries in set to GSI 'FT' - add attribute data to ElasticSearch for full text searching 'FTg' - add attribute data to GSI and ElasticSearch
IncP	Include Attribute in propagation	String Set	Yes	

Type Mappings

Type mappings define the match between the GoGraph type and its associated database type.

Load RDF data

Source	
RDF File: _:blankNode *__type* Person _:blankNode *Attr-of-Person* <value> …	All attributes of Person (node Type) defined in Type Definition.
Type definition: Person <shortName> Person <List attributes: specifying GG type (see Ty above)> …	
Write NV and propagated data -> DB db.SaveRDFNode( ) ?.PropagateScalar() maps GoGraph Type to database Types.	Maps: *GoGraph* Type -> DB Type

Query Data

Source
Type definition: Person <shortName> Person <List attributes: specifying GG type (see Ty above)> …

Load DB data -> Block Cache
 	
Pkg: db.FetchNode
 	
`data := make(blk.NodeBlock, *result.Count)`
`err = dynamodbattribute.UnmarshalListOfMaps(result.Items, &data)`

Pkg: block Define DB attribute with: type DataItem struct { . . . } Define Access Functions with : block.*Get<GG Type>()* map DB type to GG Type.
Block Cache -> ds.ClientNV Maps GG Type to DB attributes via Get<GG Type>() functions Pkg: ds, type ClientNV []*NV func (nc *NodeCache) UnmarshalNodeCache(nv ds.ClientNV, ty_ ...string) error *Unmarshal relies upon block.Get<GG Type> functions*
Execute Query uses ds.ClientNV

RDF Data

Examples of graph data stored in RDF (Resource Description Framework) format can be found using the URL:

Github: https://github.com/rosshpayne/GoGraph/data

The *GoGraph's* data loader only accepts RDF in the format described in the next section.

RDF Specification

The format for an RDF triple is as follows:

<subject> <predicate> <object> .

Comments may follow the dot (".") in the usual form of // or /* */

RDF Triple Component	Contents	
subject	A *blank node id* in the form: _:<*userDefinedString*> *userDefinedString* should match up with another triple with predicate *__type* defined.	
predicate	Matches an attribute name in a user defined type, or one of the system predicates describe below. A scalar-predicate matches a scalar attribute in a type definition. A UID-predicate matches a nested-type attribute in a type definition. A UID-predicates is internally represented by an array of UUIDs	
	System Predicates	**Description**
	__type "<*type name*>"	Specifies the Node's Type which must be defined in Type Table.
	__ID <*UUID*>	Explicitly defined node *UUID,* which is helpful for testing purposes. If not used *GoGraph* will generate a UUID during load.
object	Either a *blank node id* in the case of a UID-predicate, or a *literal scalar value,* when the predicate matches an attribute name from the node's type definition (from type system) that is a scalar type.	

Example of RDF Data: Relationship Graph

The RDF data that was used to populate the Relationship Graph is presented below. Note, there are only five *Person* nodes defined. The limited data set is intentional as its main objective is to validate the functional test cases which is best achieved when the data is simple and well understood.

```
_:abc __type "Person" . /* specify type for blank node */
_:abc __ID "66PNdV1TSKOpDRlO71+Aow==" .
_:abc Name "Ross H Fullerton" .
_:abc DOB "13 March 1958" .
_:abc Age "62" .
_:abc Siblings _:b  .
_:abc Friends _:d .
_:abc Friends _:c .
_:abc Siblings _:c  .
_:abc Cars "Fiat"  .
_:abc Cars "Honda"  .
_:abc Cars "Alfa"   .
_:abc Address "67/55 Fauxvoue St Lombardi, Cuffi, Italy"
_:abc Comment "Another fun  video. Loved it my Grandmother was from Passau. Dad was over in Germany but there was
something going on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend
everyone." .
_:abc SalaryLast3Year "112000" .
_:abc SalaryLast3Year "152000" .
_:abc SalaryLast3Year "218000" .

_:b __type "Person" . /* specify type for blank node */
_:b __ID "TsI2Qay8T2SxnOcuI3pPeg==" .
_:b Name "John Fullerton" .
_:b DOB "2 June 1960"  .
_:b Age "58"  .
_:b Siblings _:abc  .
_:b Siblings _:c  .
_:b Cars "Suzuki"  .  // specify type for blank node
_:b Cars "Honda"  .
_:b Cars "VW Golf"   .
_:b Friends _:abc .
```

```
_:b Friends _:c .
_:b Comment "A foggy snowy morning lit with sodium lamps is an absolute dream" .
_:b SalaryLast3Year "100000" .
_:b SalaryLast3Year "122000" .
_:b SalaryLast3Year "145000" .

_:c __type "Person" . /* specify type for blank node */
_:c __ID "6KIuWuyTRKSgDqwYAghl7A==" .
_:c Name "Ian Fullerton"  .
_:c DOB "29 Jan 1953"  .
_:c Age "67" .
_:c Siblings _:abc  .
_:c Siblings _:b    .
_:c Friends _:abc .
_:c Friends _:b .
_:c Friends _:d .
_:c Cars "VW Passat"  .
_:c Cars "Mitsubishi"  .
_:c Cars "Ford Laser"   .
_:c Cars "Honda"   .
_:c Comment "One of the best cab rides I have seen to date!?" .
_:c SalaryLast3Year "90000" .
_:c SalaryLast3Year "90000" .
_:c SalaryLast3Year "110000" .

_:d __type "Person" . /* specify type for blank Payne node */
_:d __ID "OlTBKXemTNWATwDDt6U5/A==" .
_:d Name "Phil Smith"  .
_:d Siblings _:e    .
_:d Friends _:abc .
_:d Friends _:b .
_:d Friends _:c .
_:d DOB "17 June 1976"  .
_:d Age "36" .
_:d Cars "Roll Royce"  .
_:d Cars "Bentley"   .
_:d Cars "Honda"   .
_:d Cars "VW Golf"  .
_:d Comment "It seems to me the camera's Smith focus is better, clearer, thank you for sharing, I think Austria is
a beautiful country I'm not surprised that it produced so many great classical musicians." .

_:e __type "Person" . /* specify type for blank node */
_:e __ID "6nG/Cd+dSoyD48CrXQjrLQ==" .
_:e Name "Jenny Jones"  .
_:e Friends _:abc .
_:e Friends _:b .
_:e DOB "17 February 1963"  .
_:e Age "59" .
_:e Cars "Renault 302"  .
_:e Cars "Zonda"  .
_:e  Comment "A great video Payne which I have enjoyed watching. It is always very interesting to see such
beautiful scenery which I am unable to visit personally. Thank you." .
```

A Concurrent RDF Loader Design

Github: https://github.com/rosshpayne/GoGraph/rdf

Let's write a Go program to load a RDF file, like the Relationship RDF file displayed in the previous section, into the *Dynamodb* graph table. To make it more challenging, let's presume the RDF file is tens of MBs if not GBs in size[8]. To expedite the load we need the program to consume all of the available compute resources (think cores not servers) and database IO throughput (think distributed database at massive scale e.g. *Dynamodb*).

This leads us to Go's best feature, in the author's opinion, which is the adoption of *communicating sequential processors* (CSP) as a model for concurrency[9]. In Go CSP programming is native to the language and consequently is as accessible as any other other language construct providing the ability to scale an application across cores with relative ease. It enables the construction of complex communicating concurrent processes with little concern for the mechanics. There is however one gotcha that needs to be kept in mind - shared variables - which exists between concurrent routines and the potential they present for data corruption due to race conditions. Fortunately the language provides a rich set of APIs to manage this issue and a toolset to verify the code as "concurrency safe". Once this is understood CSP programming becomes the default way to approach any design rather than as the exception.

At this point if you are unfamiliar or need a quick reminder of the Go CSP semantics and support packages please review Appendix A.

To achieve this scale out capability the program designer first deconstructs the problem into a number of tasks and decides which tasks can run concurrently, either as parallel routines (i.e. as multiple concurrent versions of itself) or as a single asynchronous routine that may be pipelined together with other concurrent tasks.

A quick list of potential candidate concurrent routines are:

Candidate Concurrent Routine
Read RDF file into batches of triples.
Validate RDF tuple
Accumulate Graph Edges
Convert internal IDs to Universal Unique Identifiers (UUIDs)
Error logging
Event and Stats logging
Save each valid RDF triple representation to the database.
Create a graph edge (attach a node to a node) and persist its representation to the database
Manage the instantiation of parallel concurrent routines

Not surprisingl, the concurrent Loader program makes extensive use of CSP. Run the Loader on a CPU with more cores and the Go runtime scheduler will dynamically allocate the concurrent routines to use as many cores possible resulting in a faster load.

The objective is to enable the program to scale so as to reach the limits of the database throughput, which in the case *Dynamodb* can be exceptionally high.

Concurrent *goroutines* fall into two categories. One type provides a long running *service* to the main program (aka "*main goroutine"*) or to other *goroutines* and performs a relatively simple task, like serialising access to a resource with some

---

[8] The limit on file size is approximately half the available memory on the server as the entire file is ingested into Go memory structures during the load and validated in memory

[9] The author can think of only two other languages that support CSP natively, namely, ES6 and Oracle's PL/SQL via the *dbms_alert* package. Python probably has a library that achieves some level of CSP programming but that is not native to the language.

data enhancement, and uses *channels* to asynchronously respond to and communicate with other *goroutines*. *Services* are typically started by the main program and are shutdown when the main program terminates. For example the *Error logging* and *UUID Generator* routines would each run as a *service.* A service goroutine typically provides serialised accesss to a shared resource

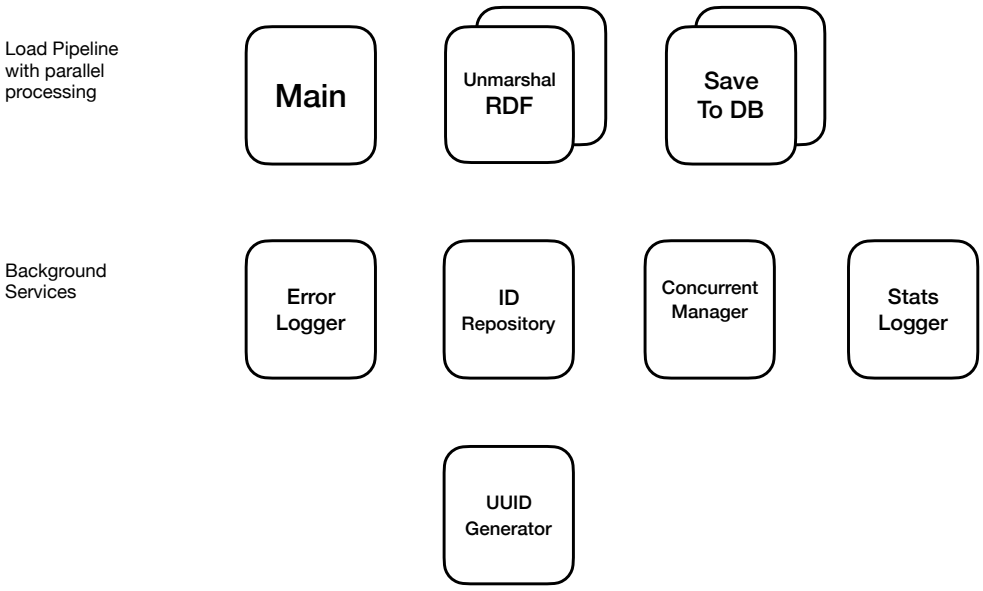
Other *goroutines* are designed to run in parallel, usually performing heavy workloads that require some scale out capability. Routines *Validate-RDF-triple* and *Save-RDF-triple* would be prime candidates to run as parallel goroutines. To support the instantiation of parallel routines the Loader makes use of a *service* that manages the instantiation of these parallel routines*,* called the *Parallel Manager Service*.

Finally, *goroutines* can be assembled into **pipelines**, where they are chained together and the output of one *goroutine* becomes the the input to the next *goroutine* in the chain. Channels are used to synchronise and pass information between the routines. The Loader program has chained the following *goroutines* together into a pipeline for end-to-end processing of RDF triples, from input file to database.

Main[10](read RDF input) | *UnmarshalRDF* | *Save-to-DB*

The Unix pipe symbol | represents Go *channels.*

From the list of candidate concurrent routines we can allocate them as follows:



Not shown in the above diagram are the associated *channels* that connect the *goroutines* together, either to exchange data and/or synchronise with one another.

The *Save-to-DB goroutine* first saves all the node data to the database and then runs the Attach-Node operation which logically attaches a child node to a parent node, for each edge defined in the RDF data. The attach-node is itself run as two concurrent *goroutines.* One routine reads the parent node data from the database (a multi-millisecond operation), concurrently with another *goroutine* that reads the child node data (another multi-millisecond operation) and then connects the child to the parent node by adding the associated “connect” items to the database under the parent node UUID. Information is exchanged via a single channel while the two *goroutines* are synchronised at various points using both a *channel* and the *sync package (ie. methods on type Waitgroup*).

Not ACID Compliant

While *Dynamodb* provides limited ACID compliance via the *TransactionWriteItems* & *TransactionGetItems API's,* as it is restricted to no more than 25 items involved in a transaction. Consequently Dynamodb transactions will not support GoGraph types with more than 23 scalar attributes as each attribute will be represented as a single item when propagated to a parent node, during the *scalar propagation* phase of the load process. Designing, coding and testing a fully transactional load program would be a non-trivial exercise so it has been left as a potential future enhancement.

---

[10] The main *goroutine* has several responsibilities but as far as the pipeline is concerned it reads RDF tuples into batches, writes a reference to the batch to a channel which is then read by the next routine in the pipeline.

Load Performance

To process 1.1 million triples in the Movie.rdf file took 1hr 18min for a concurrent setting of 16 on a *m5.large* EC2 instance (2 vCUPs, 8MB). This is much slower than expected. The cause was identified as a design oversight with a node's *Reverse Edge* attribute. The design uses a Binary Set datatype to store the reverse edge data for each edge, which consists of a parent node's UUID and its Edge Predicate's short name, to which the child node is attached. In the case of the genre node, which can be attached to thousands of parent nodes, this presents a couple of problems. The first problem is data size, as the 400KB item size limit will ultimately be reached on a large graph wth "type" nodes (like Genre). The other problem is adding a new reverse edge to a Binary Set progressively slows as the Binary Set grows in size, to the point where it is taking around 2 seconds to update a reverse edge for some Genres, like "Drama", which is a very popular genre. While the solution exists in *GoGraph* to fix both problems it was not applied to Reverse Edge attributes, only to Edge Predicates (to fix the million subscriber problem). This was a complete oversight. I expect that if it had been applied to the Reverse Edge attribute then the load time would be reduced by around 20% to 30%.

Why didn't I fix the Reverse Edge issue and reload the data? GoGraph development had reached its "use by date". It is a personal project, not a product that I expect people to use, although I built it with that in mind. I have identified the problem, I have a solution that is working on other attributes, that is sufficient. Time to move onto the next project.

Storage Abstractions

An understanding of the Dynamodb's storage hierarchy will help in understanding the meta-data design implemented in *GoGraph's* Sort Key values, and UID-overflow design.

The top component is not surprisingly a *Dynamodb* **table**. All graph data, for all graphs, is stored in a single table. The table's primary key is a composite of a binary partition key and a sort key of type string. A *Dynamodb* table is split into separate **storage partitions** (SP), each allocated to a different server as a way to spread the data and database load over multiple compute and storage resources. The number of SPs is determined by the table's read and write capacity where each SP is constrained to a maximum of 3000 read capacity units (RCU) and 1000 write capacity units (WCU). RCU and WCU are used to scale the throughput of a *Dynamodb* table and also constraint its associated read/write costs.

The table's partition key stores the UUID of either a **node block** or its associated **overflow blocks** if any. A **block**[11] is an ordered collection of one or more **items** (another *Dynamodb* abstraction) sorted by the item's sort key value. Block's provide the facility, via the Query API, for the database to filter data at the storage level rather than return data to the application to be filtered, improving query latency and minimising database read costs.

The table below summaries the various hierarchy of storage abstractions.

<table>
<tr><td rowspan="4">Table (Dynamodb abstraction)

A single key-value store for all graph data

Max Size: unlimited</td><td colspan="3"></td></tr>
<tr><td rowspan="3">Storage Partition (Dynamodb abstraction)

Internal to Dynamodb and a fundamental IO scaling component. Dynamodb partitions a table into one or more SPs the number of which is based on the table's IO capacity. Items are assigned to SP hashed on partition key value.

Dynamodb replicates each SP across multiple Availability Zones for redundancy which can also aid in faster query response.

Max Size: 10GB
Number: dynamic</td><td colspan="2"></td></tr>
<tr><td>Block (GoGraph abstraction)

See Section: GoGraph can use the leading portion of the sort key to effectively partition the node data into blocks, which Dynamodb can retrieve in a single get operation rather than requiring an expensive scan operation thereby minimising query latency and IO costs.

Access by Dynamodb API Query + StartWith(SortK,:p).

Max Size: unlimited
Per Node: up to 20 blocks</td><td></td></tr>
<tr><td>Item (Dynamodb abstraction)

Each node or overflow block contains one or more Dynamodb items.

Access by Dyamodb API GetItem.

Max Size: 400K
Number: unlimited</td></tr>
</table>

Block Types

As detailed in the table below there are two types of blocks, **Node** and **Overflow**. *Node blocks* are associated with the node's meta and scalar data. *Overflow blocks* are associated with each of a node's UID-predicates (child node UUIDs). Internally, *GoGraph* partitions a *Node block* into a number of sub-blocks, such as *Node-Type, Scalar-Data, Edge-Data* - see table below for complete list. The only block that is configurable by a *GoGraph* application designer is the *Scalar-Data* block within a node block. A designer can further sub-partition a *Scalar block* with upto 5 more partitions (aka sub-blocks) as required. All other sub-blocks in a Node block are internal to GoGraph and cannot be changed.

Type	Description		
Node Block	Contains the data related to a node. Every node contains the same data elements, albeit some may not be present in some nodes depending on its type, as describe in the list below;		
	Partitions	**Configued By**	**Description**
	Node Type	System	
	Scalar Data	User	As defined in the user-defined type for the node type. Scalar data is represented by a user-defined types scalar predicates.
	Edge Data	System	Optional. Edge data denotes the collection of node-node edges for a particular *Edge Predicate.* Each *Edge Predicate*(1) has its own edge data which internally is represented by a Dynamodb *List* data type constrained to hold *UUIDs only.*
	Propagated Child's Scalar data	System	Optional. For each UID type in the nodes user-defined type there will be data elements for the each child node's scalar data (if defined).
	Double Propagated Scalar data	System	Optional. For each UID type in the nodes user-defined type there will be data elements for the each grandchild node's scalar data (if defined). This is only applicable when the grandchild has a 1:1 cardinality with its parent type.
	Reverse edges	System	Optional. While a graph's edge is unidirectional (parent->child) it is useful to store the reverse edge (child->parent).
Overflow Block	Optional. System configured. Overflow blocks are allocated on demand as more child nodes are attached to a parent node. There is no restriction on the number of nodes that can be attached to a parent node. See Section Overflow Block Design for further details.		
	Partitions	**Configued By**	**Description**
	Edge Data	System	Optional. Edge data denotes the collection of node-node edges for a particular *Edge Predicate.* Each *Edge Predicate*(1) has its own edge data which internally is represented by a Dynamodb *List* data type constrained to hold *UUIDs only.*
	Propagated Child's Scalar data	System	Optional. For each UID type in the nodes user-defined type there will be data elements for the each child node's scalar data (if defined).
	Double Propagated Scalar data	System	Optional. For each UID type in the nodes user-defined type there will be data elements for the each grandchild node's scalar data (if defined). This is only applicable when the grandchild has a 1:1 cardinality with its parent type.

Design Highlights

Research has found the most significant factor in determining a database's query response is the degree to which data is clustered at the storage level. Documented below are the major design decisions that were made to maximise data clustering or more generally aid in faster query performance. Broadly these design decisions can be categorised into **data duplication**, **attribute compression, node data partitioning, table key design** and **GSI design.**

As previously mentioned, GoGraph makes use of a single table to store all the graph data. The table and its associated GSI's are defined in this JSON script:

Table create JSON: https://github.com/rosshpayne/GoGraph/json/dgraph-table.7.json

Design Decision	Clustering Benefit
Single Table	All graph data, for all graphs, is stored in a single table in *Dynamodb*. There are separate tables for *type* data and logging.
Composite Primary Key	The single graph table has a composite primary key. This forces clustering of data around the *partition key* value and guarantees the data is sorted within the partition key based on the *sort key* value. The data can be further partitioned within a partition key using an appropriate sort key design. See *Sort Key Design*
Node's UUID as the partition key	Using a node's *UUID* as the *partition key* means a node's *scalar* data and other node related data (*UID predicates*, *propagated* data etc) is clustered under the one index entry and can therefore be accessed via a single *Dynamodb query*. The data indexed under a node's *UUID* is referred to as a **node block**.
OverFlow blocks	Having all the node's data stored under one *partition key* value means scalable read options are limited as it will ultimately lead to IO contention on the associated storage partition. To circumvent this it is possible to configure a node to have overflow blocks which extends the node data across multiple partition keys when a configured data size is reached. While the data is still clustered within each partition key, multiple partitions means the option of concurrent reads without risk of IO contention is now possible. See Section ?.
Sort Key Design	A thoughtful *sort key* design enables partitioning of a node's data into commonly accessed items e.g. *scalar* data, *propagated scalar* for each *UID predicate or all the node data.* By specifying the appropriate *sort key* string a query has some control over just what is returned to the application potentially saving significant number of RCUs. See Section ?
Sort Key compression	To increase the clustering factor any attribute name referenced in the *sort key* uses the attribute's short name. Note: *as part of a type definition each attribute is assigned a short name which is typically one character. An attribute's short name must be unique to the type in which the attribute belongs.* Given some sort keys refer to multiple attribute names this can save considerable storage and reduce the amount of data returned to the application which may result in less database calls.

Design Decision	Clustering Benefit
Partitioning node data for optimised reads	The Sort Key specification provides the data designer with up to three levels of nesting, enabling a query to target subsets of node data by varying the leading portion of a Sort Key value stipulated in the query definition. The flexibility to tailor a query using the Sort Key value prevents unwanted node data from being returned to the application, minimising response times and Dynamodb read and network costs. The benefits of partitioning increases as the quantity of data stored in a node increases. The top two levels of nesting divides the data into **node data** and **reverse edge data**. Within the node data partition, the Sort Key specification can be used to further partition the node data into **scalar**, and **propagated** data. Within the *scalar* partition, the data designer has the option to sub-partition it into a further 6 partitions, while the *propagated* data can be further partitioned into up to 5 sub-partitions.
Duplicate data to improve data clustering	An important part of the GoGraph design is the duplication of a node's scalar data into its parent node. This feature is called **scalar propagation** and is carried by the *attach-node program***.** This feature enables the clustering of all sibling node data into a single block in the parent node, enabling efficient querying of all child node's scalar data. Instead of one IO per child node, scalar propagation enables hundreds or even thousands of child nodes to be queried in a single read operation. Data propagation effectively reduces the depth of a graph from *n* to *n-1, which has huge IO benefits particularly for highly connected graphs.* Further to the design, for the particular case where a child node's *UID predicate* has a 1:1 cardinality with its child node, the scalar data of each grand-child node can be duplicated all the way up to the grandparent node, effectively reducing the depth of a graph from n to n-2. *There is of course a cost to this design in maintaining the data consistency on updates to a node's scalar data or the delete of a node, making sure the updates or deletes are reflected in the parent node. I have conveniently ignored this problem for the purpose of benchmarking the query performance of GoGraph, but it would appear to be a good use case for Dynamodb streams which could handle all the data consistency issues asynchronously.*
List Data Types for Columnar like querying	The propagated scalar data is stored in *Dynamodb's* **List** datatypes in the parent or grandparent node. See the Data Model section for further details. Each newly attached node will have each one of its scalar data predicates appended to a List datatype appropriate for the predicate's type. The List type is very compact form of data storage that lends itself to fast in memory and storage scans. In essence it is very similar to Columnar format found in many databases technologies but in this case is a by product of the GoGraph storage design.
Gobal Secondary Indexes	The GraphQL root queries will make use of GSI's to retrieve the initial set of candidate node UUIDs. Each scalar attribute has its own sparsely populated GSI. There is also a GSI for the type of the node, to help resolve root queries based on type. The GSI json definitions is include in the table definition link at the top of this section.

The Million Subscribers Problem

Some YouTube content creators have millions of subscribers. How would *GoGraph* handle data volumes at this scale, while avoiding the potential of severe contention on the parent-node block?

The short answer is *GoGraph* assigns edge data their own partition keys and hence the two types of data, ***scalar*** and ***edge***, are unlikely to contend for the same *Dynamodb* resources. In fact, the more edges (subscribers) that are added to a parent node, the more partition key values will be assigned and the greater the potential for more storage servers to be used, leading to less resource contention than more. So distributing edge data across more database partitions as the number of subscribers grows is how *GoGraph* solves the million subscriber problem. However splitting data across partitions reduces the data clustering factor which will disadvantage queries. Lets drill into the design a little more and see how distributing edge data can in fact advantage queries.

Firstly, the first twenty edges are stored in the node's edge attribute (identified by its unique sort key value - see next section for more details) in the data-block. *Edge attributes* are also known as *UID-predicates* because they only contain UUID data, one child UUID for each parent-child edge. However, all subsequent edges will be added to an *Overflow Block* (OvB) assigned its own UUID, rather than the edge attribute in the parent's data-block. The UUID of the OvB is added to the parent node's edge attribute instead. A parent node can support up to ten OvBs per edge attribute, which are dynamically created as more edges are added, thereby spreading the edge data across more Dynamodb partitions.

GoGraph will flag each UUID in the edge predicate of the parent node, as either a *child node* or an *OvB* UUID.

How UUIDs are stored in an OvB?

As previously mentioned, the only data that is required to define a graph's edge for a particular UID-predicate is the UUID of the child node. Because the edge data is stored against the parent node, we immediately have the the two UUIDs necessary to define an edge - all accessible in one IO from the parent-node data block. Until, that is, more than twenty edges have been saved, at which point the child node UUIDs will be stored in a OvB.

OvB store the edge UUIDs using the same design employed by all *UID-predicates* in the parend-node block. All UUIDs are stored in a *Dynamodb* ***List*** data type. Significantly, a *List* data type guarantees the order of the UUIDs is maintained. So the 50th edge added to a parent-node will always be the 30th entry in the List type of the first item of the first OvB created. Similarly, the 5000th edge will always be entry 4970 in the List type of the first item in the first OvB created. How this magic is maintained has a lot to do with the Sort Key design discussed in the next section.

List types also aid in compression of the edge data. There is no overhead in storing an edge, only the UUID of the child node as an OvB imposes next to no overhead. It also aids in querying speed as a List type emulates a Columnar type found in databases that use this type to speed up queries on large data sets.

However, at some point, as more edges are added *GoGraph* will hit the *Dynamodb* imposed 400KB item size limit. *GoGraph* will use this as a trigger to create of a new OvB and allocate a new item to the OvB with an empty List type. The new edge will then be allocated to this empty List. This pattern will continue until *GoGraph* has created ten OvBs, at which point any new edge will be randomly allocated to one of the existing OvBs.

Given a UUID requires 16 bytes of storage, approximately 25,600 edges can be stored in one *Dynamodb* List type provided it represents the only non-key attribute in the item. To store 1 million edges (aka subscribers) would require 40 items which *GoGraph* would have distributed over ten OvBs. In terms of *Dynamodb* table partitions, the edge data has been spread over ten partitions plus one for the parent node's data block.

The same OvBs are used to store the results of *scalar propagation* of each child node. Each scalar attribute will be represented as an item in the OvB identified by its Sort Key value. The item will contain a suitable List type to store the scalar data from each child node. The order of the scalar data in the List type matches the child UUID order in the List type containing the edge data, which is essential otherwise it would not be possible to map the scalar data to the relevant child node. *Scalar propagation* exists purely to speed up graph queries, as the parent and child data has been effectively clustered together reducing the fetches required to resolve certain queries involving the child nodes.

While not yet implemented the OvB design could also be used to store **reverse-edge** data**,** i.e. the child-to-parent edges, which is another potentially big predicate data wise.

A more detailed description is delayed until the GoGraph Data Model is discussed on Page 25

Table Key Design

GoGraph stores all graph data in a single table in the case of Dynamodb and four tables in the case of Google's Spanner. In both cases the type information is stored in a table separate from the graph data.

There are four categories of graph data.

Category	Associated block	Description
Scalar	Node's data block	all the scalar attributes associated with the node's type
Edge	Parent node's data block plus one or more Overflow Blocks (OvBs)	all the child UUIDs linked to a parent node for each edge predicate. Edge data is stored in either the parent node or Overflow blocks.
Propagated Scalar	Parent node's data block plus one or more Overflow Blocks.	Scalar data associated with each child node in each edge-predicate
Double Propagated Scalar	Parent node's data block plus one or more Overflow Blocks.	Scalar data associated with each grand-child node in each edge-predicate where there is a 1:1 relationship with the grand-parent.

The table's sort key can be used to target each of the above categories as well as attributes within each category - but lets not get ahead of ourselves. The next section examines the table's key design in detail and is critical to understanding how GoGraph works.

Composite Primary Key

The single graph data table employees a composite primary key design, comprising a *partition key* and *sort key.*

Key	GoGraph Type	Table Attribute	Dynamodb Type	Size in Bytes	Description
Partition Key	UUID (1)	**PKey or PK**	Binary	16	Uniquely identifies each type of GoGraph block, ie. either a *node data* block or an *overflow* block.
Sort Key	string	**SortK or SK**	String	2-24	Uniquely identifies the categories of data within a GoGraph block. Determines the non-key attributes to expect. See Sort Key Specification.

1. Universally Unique Identifier (UUID) version 4, a 128-bit number

Each instance of a block (*node* or *overflow*) is identified by a UUID. Data within a block is targeted using the sort key value.

Sort Key Description

Fundamental to the *Sort Key* design are *partition identifiers,* assigned in the type system to each node attribute (see Table: ?) and enables a GoGraph query to target a subsets of node data, when used in conjunction with Dynamodb's *BeginsWith* function. Partition identifiers physically splits, at the database storage level, the node data into partitions which can then be queried separately, to the rest of the node data.

Typical targets for a query that are facilitated by partition keys:

Targets for a sort key value
a single scalar attribute
a subset of scalar attributes that are commonly queried together
all scalar attributes

Targets for a sort key value
Edge data: UUID of each child node associated with a single edge predicate
Edge data: UUID of each child node associated with a single edge predicate in the parent node, that meets a particular scalar value in the child node or grand child node in the case of a 1:1 relationship between parent and grand-child node.
all scalar data or a subset of scalar data of each child node associated with a single edge predicate in the parent node, that satisfies a condition in the value of a scalar value for a child node or grand child node in the case of a 1:1 relationship between parent and grand-child node.

The table below lists the partition types and their associated values. At the top level is the *user partition*, which has the fixed value of *A,* and is used to separate the user defined data from the system generated data (which have hidden partition identifiers other than *A)*. System generated data includes the reverse edges (parent UUID) for nodes that represent a child edge for example. Queries that only reference the user partition, in the sort key value, will return or process all attributes.

The next level of partitioning is defined for the scalar and edge attributes of a node. Up to six *scalar* and up to five *edge* partitions are available. The default is a single partition for each.

The ability to physically partition the data is very useful when it is disparate data of varying volumes and most queries target only a small subset of the data each time. On the other hand, if the data is not particularly disparate and always queried as a whole most of the time, partitioning scalar or edge attributes beyond the default is not really warranted.

The principal advantage of partitioning node data is that it gives the designer the capability to write queries that minimise both read latency and the associated cloud costs (aka Read-Capacity-Units or RCUs).

System Partitions	Sort Key	Other attributes	
Node Scalar and Edge Data	*A#*		
Node Type (single item for each node)	*T#*		
Reverse Edge	*R#*	*Lpuid - List of parent UUID* *LSK - List of SK* *LOvBid - List of OvB* *LOvBbid - List of Batch Id* *LibId - item in batch*	*Quick access to associated parent edge - useful during soft delete operation.* *Overflow block design applies*

Node Partitions	Partition Identifiers*	Default Partition	Description
Scalar	A..F	A	Attribute to partition mapping defined in attribute type definition. Separate scalar predicates into different partitions. For example if there are tens of scalar predicates then it may make sense to group them into a few partitions based on predicates that are commonly queried together.
Edge	G..K	G	The node's edge data (parent to child nodes for each edge predicate) can be separated into five partitions. (Currently only one partition, G, is supported)

* as specified in the associated attribute's type definition

Sort Key Specification

To keep the sort key as short as possible only short names are used for all predicate names. A predicate's short name is defined in its associated attribute in the type definition (where predicate name matches a type's attribute name). The specification also references the data node partitions described in the table above.

Node Data	Sort Key Specification
Scalar	*<systemPartition>#<ScalarPartition>#:<ScalarPredicateShortName>* *Note: the default for each scalar attribute is systemPartition "A" and ScalarPartition"A"* *Examples of the leading portion of a Sort Key value when passed into Dynamodb's BeginsWith function to target one or more attributes:.* ***A#*** *// all attributes* ***A#A#*** *// all attributes in scalar partition A* ***A#F#*** *// all attributes in scalar partition H* ***A#A#:S*** *// attribute Serial Number (short name S)* ***A#B#:Rv*** *// attribute Revenue (short name Rv)*
Edge *Overflow Block variant*	*<systemPartition>#<EdgePartition>#:<EdgePredicateShortName>* *Examples of the leading portion of a Sort Key value when passed into Dynamodb's BeginsWith function to target one or more attributes:.* ***A#G#*** *// all edge attributes* ***A#H#:U*** *// edge attribute UsedBy (short name)* *<systemPartition>#<EdgePartition>#:<EdgePredicateShortName>#<UIDPartition>#:<EdgePredicatehortName>#<OvBbatchID>*
Propagated scalar data *Overflow Block variant*	*<systemPartition>#<EdgePartition>#:<EdgePredicatehortName>#<ScalarPartition>#:<ScalarPredicateShortName>* ***A#G#:U#C#:A*** *// attribute A (short name) promoted scalar data for node edge predicate U (short name)* ***A#G#:U#C#:N*** *// attribute N (short name) promoted scalar data for node edge predicate U (short name)* *<systemPartition>#<EdgePartition>#:<EdgePredicatehortName>#<ScalarPartition>#:<ScalarPredicateShortName>%<OvBbatchID>* ***A#G#:S#A#:F%2*** *(A#G#:S is redundant - consider removing)*
Propagated grandchild node scalar predicates (for 1:1 cardinality only) *OvB variant*	*<systemPartition>#<EdgePartition>#:<EdgePredicatehortName>#<grandchildEScalarPartition>#:<grandchildEdgePredicatehortName>#ScalarPartitionShortName#:<childPredicateShortName>* ***m|A#G#:P#G#:A#A#:N*** ***(code needs to be updated)*** *<systemPartition>#<EdgePartition>#:<EdgePredicatehortName>#<EdgePartition>#:<grandchildEdgePredicatehortName>#ScalarPartitionShortName#:<childPredicateShortName>#<OvBbatchID>* ***m|A#G#:P#G#:A#A#:N%1*** ***(code needs to be updated)***
Overflow Block Header	***OV***

For an example of *sort key* usage see the graph data from the Movie database at the end of the next section.

The table below lists the system generated data for each node. show the system Sort Key values that are generated for each node.

The Data Model

GoGraph translates lines of RDF into a sequence of database items each employing a unique combination of database attributes as defined by the data model presented in the following two tables.

The data model also defines performance related enhancements such as the items and their attributes that store the propagated scalar data and the reverse edge data. Table 2 describes the data stored in the Overflow Blocks (OvB).

Lets start by reviewing the node data model in Table 1.

The fist column in the table lists the types of items that describe node data, such as the scalar types, edge predicates, GoGraph set type, list types etc. For example, the database item that describe the node type contains attributes:

PK, SK, Ty

where

PK : UUID of node
SK : Sort Key value “T#”
Ty : type value (long name) of the node as defined in the type system.

To hold the scalar String value for a node predicate defined in the RDF is:

PK, SK, S, P, Ty

where

PK : UUID of node
SK : sort key value (e,g, A#A#:F) - see Sort Key Specification.
S : stores the String value
P : stores the scalar predicate name as defined in the type system and the RDF input file.
The value will also be populated into the associated global index for root queries purposes.
Ty : contains the value for String type. This populates a global index.

If the scalar predicate in the RDF is a float type then the database item would consist of the following database attributes

PK, SK, N, P, Ty

If the GoGraph type is a String List, then the database item/row would consist of:

PK, SK, LS, P, Ty

If the GoGraph attribute defines an edge ie. parent to child connection defined by the a list of child UUIDs, the associated database item/row will consist of:

PK, SK, Nd, XF, Id, P, N

It’s worthwhile explaining the associated *Dynamodb* type for *Nd, XF* and *Id* attribute is the List type. This takes advantage of the List types characteristic of guaranteeing the order of the data in the List. This means, for example, that the index entry 6 in Nd is the 6th child added to the parent node edge. Similarly the same index entry in XF, and Id represent the same child node.

By the time the 100th child node is added to the edge that database will contain one entry in the node block and one entry in the Overflow Block as only the first twenty child nodes are stored in the node block exclusively. So in the case of the 100th child node there will be two relevant database items/rows. The first will consist of the same attributes as above, namely:

PK, SK, Nd, XF, Id, P, N

but the associated Nd entry will now contain the UUID of the overflow block (OvB) not the child node block. The associated XF and Id values will be relevant to the OvB (description in second table below). The actual child UUID will be contained in a item/row in the OvB, consisting of the following attributes:

PK, SK, Nd, XF, N

where PK is the UUID of the OvB and SK will be of the form A#G#:E#G#:E%1

Node Block	Dynamodb Table Attribute	GoGraph Type	Dynamodb Type	Description
Composite Key	**PKey**	Binary	Binary	Partition key - Node's UUID
	SortK	String	String	Sort Key - see section on Sort Key Design
Node Type	**Tylx**	String	String	User defined type (short name) prefixed with graph short name e.g m|Pf **SortK value: m|T#**
	~~**IsNode**~~	~~String~~	~~String~~	~~??~~
	~~**Graph**~~			~~depreciated: graph short name~~
Scalar Types	**S**	String	String or Null	String data
	N	Int or Float	Number or Null	Rust: **Option::None** represents null value The type system determines whether the value is represented as I64 or f64 in Rust. Alternative: use AttributeValue::NULL to represent NULL in each datatype
	B	Binary	Binary or Null	Binary data
	Bl	Boolean	Boolean or Null	Boolean data
	DT	DateTime	String or Null	DateTime data
	P	String	String	[for indexing purposes]. Global indexes, P_N, P_S, P_B, use this attribute as the partition key. These indexes are used for root queries. e.g. Name = "Smith" spec: graph|attribute|node-type e.g. "m|Name|P" S attribute contains the value of the name. Each scalar attribute item must have a P attribute defined if it is to be indexed.
	~~**Nu**~~	~~Boolean~~	~~Bool~~	~~For nullable types. True - associated scalar is NULL ie. not defined. False or empty - associated scalar is not null (defined)~~
	Ty	String	String	[for indexing purposes]. User defined type (short name) as used -
	~~**TyA**~~	~~String~~		~~*Evaluating this entry.* Show what attribute cantains data.. I,F,B,Bl,Ns,Ll,LF…~~ ~~*Not sure if useful, as can determine from type system, based on sortk.*~~
Set Type	**NS**	Number Set	Number Set	Set of numbers
	SS	String Set	String Set	set of strings
	BS	Binary Set	Binary Set	Set of binary
	*PBS**	*Binary Set*	*Binary Set*	*Set of binary*
List Type	**Ll**	Number List	List	List constrained to number
	LF	Float List	List	List constrained to float64
	LS	String List	List	List constrained to string
	LB	Binary List	List	List constrained to binary

Node Block	Dynamodb Table Attribute	GoGraph Type	Dynamodb Type	Description
	LBl	Boolean List	List	List constrained to boolean
Edges **(Edge/UID Predicate)**	**Nd**	UUID List	List	Ordered list of Binary (child node UUIDs)
	XF	Int List	List	UUID type flag: 1 - *child UUID* *2 - child InUse* *3* - Child *Detached* *4 - Overflow Block UID* *5 - Overflow Block In Use* *6 - Overflow Batch Full (new Batch required)*
	Bid	Int List	List	For Overflow blocks only. Current Overflow Batch Id. Range from 1 to *n*. Otherwise 0.
	P	String		Predicate Name of edge (long name) **Used to populate GSI P_N: count number of edges**
	Cnt	Int		Count of all attached nodes (children). Note: not just entries in Nd.
	<Facet>		List<scalar>	Attributes associated with the edge facet. Scalar types supported only.
Propagated Scalar Data	**LN**	Number List	List	List constrained to number
	LS	String List	List	List constrained to string
	LB	Binary List	List	List constrained to binary
	LBl	Boolean List	List	
	LDT	DateTime List	List	
	~~**Nul**~~	~~Boolean List~~	~~List~~	*Depreciated: Nullable types only. Otherwise empty (none) TRUE means associated scalar entry in List is NULL ie. not defined* *Replace: use AttributeValue::Null in data field.*
	~~XBl~~	Boolean List		*Did represent deleted. Use XF in edge instead. This will require two reads (edge and pg item) but simplifies node deletion and saves space.*
~~**Reverse Edge (Rust uses Overflow design)**~~	**BS**	~~Binary Set~~	~~Binary Set~~	~~Set of binary~~
	~~*PBS**~~	~~*Binary Set*~~	~~*Binary Set*~~	~~*Set of binary*~~

* internal to GoGraph - do not use

Overflow Block (OvB)	Dynamodb Table Attribute	GoGraph Type	Dynamodb Type	Go Type	Description
Composite Key	**PK**	Binary	Binary	[]byte	Partition key - OvB's UUID
	SK	String	String	string	Sort Key - see section on Sort Key Design for overflow sort key format
Block Header	~~**Graph**~~	~~String~~	~~string~~		~~Graph short name.~~ *All sortk values prefixed with Graph short name*

Overflow Block (OvB)	Dynamodb Table Attribute	GoGraph Type	Dynamodb Type	Go Type	Description
	OP	Binary	Binary	[]byte	Parent node UUID
Edge Paredicate **(child node UUID)**	**Nd**	UUID List	List	[][]byte	Ordered list of Binary (child node Uuid's + Overflow Block Uuid's)
	XF	Int List	List	[]int	UUID type flag: 1 - *child UUID* 2 - *Node Deleted*
	Cnt	Int	N	N	count of children assigned to edge
	P	String	String	String	graph-sn|edge-attr-name|node-type-sn e.g **m|Film.Director|P**
	Ty	String	String	Strig	Node type e.g **m|P** (maybe redundant given P attribute)
Propagated Scalar Data	**LN**	Number List	List	[]int	List constrained to number *Vec<i64> : null represented in* ***Nul (preferred as not nullable faster as no unwrap or no Nul)*** *Vec<Option<i64>> : null rep by None*
	LS	String List	List	[]string	List constrained to string
	LB	Binary List	List	[][]byte	List constrained to binary
	LBl	Boolean List	List	[]bool	List constrained to boolean
	~~**OvB**~~	~~Uuid List~~	~~List~~	~~[][]byte~~	~~Overflow Block Uuids~~
	~~**XF**~~	~~status flag~~	~~List~~	~~[]int~~	~~Overflow Block Status flag~~

1. ASZ is an attribute in overflow blocks. Keeps count items in overflow batches

---

Handling Nulls

As defined in the type system any type may be constrained to be ***nullable*** or ***not nullable.*** A nullable value can contain a null value, while a not nullable cannot contain a null value and will generate a database error if it is attempted to be saved into the database.

For nullable scalar the associated program value will use *Option::None* to represent a NULL value. The not null value will be represented by the contents of *Option::Some* . In the database the null value is represented by *AttributeValue::NULL(true)*.

Reverse Edge Design

The reverse edge predicate stores all the parent nodes to which a node has been attached, for each Edge-Predicate. The reverse edge data resides in a single item in the node block (Sort Key value is "R#"). It is allocated its own system partition separate from the main partition (Sort Key starting with "A#") containing the node scalar and propagated data.

Testing has show the reverse edge design is a little "under thought". While the data content is sufficient to define each reverse edge (UID of the parent plus the name of the Edge-Predicate), the single item makes it both slow to append, affecting RDF load times, and expensive to query, particularly for nodes that have thousands of parents.

Refactoring the reverse edge design should include the following components:

Refactoring Component	Comment
Make it optional	Not all Edge-Predicates require the reverse edge to be recorded. Those marked as **reference** types in the data model (value bool) will not include reverse data.
Create a reverse edge item per Edge-Predicate	Item design for Reverse edge data: Attributes ———— PK : child UUID SK: **R#<NodeType>#<SK-OF-TARGET-EDGE>** Target: Parent UID or OvB [List-binary] Batch: batch id if applicable [List-number] Id : index into batch This design allows all reverse edge data across all Edge-Predicates to be read with a single database read call using a leading sort key of “R#”. List attributes are used to store target UUID (either parent node or overflow block) and Batch and Id values for the overflow values, as there can be multiple instance of a parent node type references the same single child node.
Reverse Edge Overflow Blocks (REOB)	A node may have thousands or hundreds of thousands of reverse edges, which will have scalability issues if all the edges are held in one Dynamodb item. This can be readily overcome by applying the million-subscriber-design to the reverse edges as well. This will spread reverse edges across data partitions enabling parallel processing with low resource contention. This means each reverse edge item in the node will have an **XF** and **Id** attribute. REOBs will be created dynamically. The REOB will make use of the overflow blocks of the parent edge

More Details about the UID Overflow Design

Now that the Data Model has been revealed we can finish the discussion on the Overflow Block design.

As a reminder, Overflow Blocks (OvB) were introduced to enable edge data and their associated propagated scalar data to be located in a separate block to the node data, by assigning the OvB its own UUID. As more edges are added to the parent node edge the systems scales by creating more OvBs and adding the edge data to it, thereby distributing the edge data over more storage servers. So that is the theory, lets look at the implementation.

GoGraph will create an OvB when the number of child nodes for an edge exceeds twenty. As more child nodes are added up to a further nine OvBs will be created. Once this limit is reached each new child will have its details added to one of the OvBs selected on a random basis.

Each edge predicate in a node data block will have one database item containing the following attributes:

PK, SK, Nd, XF, Id

As described in Table ?, *Nd, XF* and *Id* attributes are Dynamodb *List* types. The *Nd* attribute will contain the UUID of the first twenty child nodes attached to an edge, along with identifiers in the *XF* and *Id* attributes that indicate the associated UUID is that of a child node. When an OvB is created GoGraph will append its UUID to the **Nd** attribute, and append a value 3 to the **XF** attribute and a value of 1 to the **Id** attribute, to indicate that it is an OvB not a child node.

A *Dynamodb* item in an OvB contains the following attributes:

PK, SK, Nd, XF

Again, the *Nd* and *XF* attributes are Dynamodb *List* types. There is no *Id* attribute required in the OvB - more about this later. The process of attaching a child node is now only recorded in the OvB. Each child UUID is appended to the *Nd* attribute along with the value 3 to the *XF* attribute. The value of the SK will follow the specification described in section: ?.

A Dynamodb item can be up to 400KB in size and this would accommodate approximately 25,500 child nodes, however the impact of appending each child node to the list types, *Nd* and *XF* becomes a significant performance overhead at some point as each append operation involves reading the entire contents of the List type each time to determine the end of the list. Consequently, GoGraph imposes a limit of 300 nodes or approximately 5KB on an OvB database item. Each item is given an integer identifier starting at 1 for the first item in the OvB and incrementing by 1 for each additional item created. The integer identifier of the current item (i.e. the last item created) is recorded in the *Id* attribute of the edge item in the parent node block for the associated OvB entry. The integer identifier is also appended to the SK value of that item. As a result GoGraph knows for each OvB associated with an edge what is the “load” item that it can add a new child node data to.

While there is essentially no limit to the number of items that can be created in an OvB having all child node data within one OvB has the potential for serious resource contention when either adding a lot of child nodes at once or when it comes to querying the child nodes. As a result after the tenth item has been created in an OvB, *GoGraph* will create a new OvB, add its UUID to the edge attributes in the parent node block (along with the XF & Id entries) and start adding new child nodes to it. This will continue until the tenth OvB have been created at which point *GoGraph* will start reusing the OvBs.

and incrementing identified in the SK value by a “batch” id, a value of 1 to n. When an OvB item reaches this limit a new item is created in the OvB and the value of the *Id* attribute is incremented to indicate the new item has been created. The *Id* attribute in the node data block simply tells GoGraph what is the current item in the associated OvB.

Data Example: Person Node from Movie Graph

Types

Review the following type definitions from the Movie database.

Type	Attribute (The predicate in S-P-O)	Attribute Short Name	GoGraph Type
Film	title	N	**String**
	init_release_date	R	**DT**
	film.performance	P	**[Performance]** Note: [ ] so cardinality 1:M
	film.director	D	**[Person]** Note: [ ] so cardinality 1:M
	film.genre	G	**[Genre]** Note: [ ] so cardinality 1:M
Genre	name	N	**String**
Person	name	N	**String**
	actor.performance	A	**[Performance]** Note: [ ] so cardinality 1:M
	director.film	D	**[Film]** Note: [ ] so cardinality 1:M
Character	name	N	**String**
Performance	performance.character	C	**Character** Note: no [ ] so cardinality 1:1
	performance.actor	A	**Person** Note: no [ ] so cardinality 1:1
	performance.film	F	**Film** Note: no [ ] so cardinality 1:1

Graph Data for a Person Node

Instance of **Person** scalar item. Please review the SortK Specification on page 17 for an explanation of the SortK values.

PKey	SortK	Non-Key Data: **Attribute -** Value
7kRfp8KyQ/erDWfM15ZnwA==	A#A#:N [*name*]	**P** - name **S** - “Stanley Kubrick”

Instances of Person ***Edge Predicates*** (edges), *Actor.Performance* and *Director.Film*.

PKey	SortK	Non-Key Data: Attribute - Value
7kRfp8KyQ/erDWfM15ZnwA==	A#G#:A [*actor.performance*]	**Nd** - [ZILM/pmeTTKf1oahnGYgiA== OdRS8o9eRdKnKVUA9KHStw==] **XF** - [1 , 1] **Id** - [0 , 0]
	A#G#:D [*director.film*]	**Nd** - [3WdQPRAiRG29iGuGHKuLMA== ymanEWWqSFSml9xvZK9tsA== . . . QZTfWCUWSr++82yW2QaoBQ== . . .] **XF** - [1 , 1, ..., 1, ...] **Id** - [0 , 0, ..., 0, ...]

Instances of Person, **propagated child scalar** Items for Edge Predicate*, Director.Film*.

PKey	SortK	Non-Key Data: Attribute - Value
7kRfp8KyQ/erDWfM15ZnwA==	A#G#:D#:N [title]	**LS** - [“2001: A Space Odyssey” “Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb” . . . “Eyes Wide Shut” . . .] **XBl** - [False, False, ..., False,...]
	A#G#:D#:R [initial_release_date]	**LS** - [“1968-04-02T00:00:00Z” “1964-01-29T00:00:00Z” . . . “1999-07-13T00:00:00Z” . . .] **XBl** - [False, False, ..., False,...]

Instances of Person, **propagated grandchild scalar** Items for Edge Predicate, *Actor.Performance*

PKey	SortK	Non-Key Data: Attribute - Value
7kRfp8KyQ/erDWfM15ZnwA==	A#G#:A#G#:A [performance]	**Nd** - [7kRfp8KyQ/erDWfM15ZnwA== 7kRfp8KyQ/erDWfM15ZnwA==] **XF** - [1 , 1] **Id** - [0 , 0]
	A#G#:A#G#:A#:N [performance.actor]	**LS** - [“Stanley Kubrick” “Stanley Kubrick”] **XBl** - [False, False]
	A#G#:A#G#:C [performance.character]	**Nd** - [HmarwkocQkeBsMfQaAr5xg== tZJl2ecaTDOn0NYPhcudGA==] **XF** - [1 , 1] **Id** - [0 , 0]
	A#G#:A#G#:C#:N [name]	**LS** - [“Murphy” “Bearded Cafe Patron”] **XBl** - [False, False]
	A#G#:A#G#:F [performance.film]	**Nd** - [iTjslAP3SimyDhKzQ4HZ0A== OL1MDBTnR3qltV5RCJCHuQ==] **XF** - [1 , 1] **Id** - [0 , 0]
	A#G#:A#G#:F#:N [name]	**LS** - ["Full Metal Jacket” “Eyes Wide Shut”] **XBl** - [False, False]

PKey	SortK	Non-Key Data: Attribute - Value
	A#G#:A#G#:F#:R [initial_release_date]	**LS** - [“1987-06-17T00:00:00Z” “1999-07-13T00:00:00Z”] **XBI** - [False, False] **Id** - [0 , 0]

Finally, the system generated items:

PKey	SortK	Non-Key Data: Attribute - Value
7kRfp8KyQ/erDWfM15ZnwA==	T#	**Ty** - “P” (type short name) **Ix** - “Y”
	R#	**BS** - [AJchPqDKT+a+wfnqQI9eAACXIT6gyk/mvsH56kCPXgBEIzA= OL1MDBTnR3qltV5RCJCHuTi9TAwU50d6pbVeUQiQh7IEIzA= . . . 92A+y9KHS8ePq00cUfo/zvdgPsvSh0vHj6tNHFH6P85EIzA=] **PBS** - [AJchPqDKT+a+wfnqQI9eAEQ= OL1MDBTnR3qltV5RCJCHuUQ= . . . 92A+y9KHS8ePq00cUfo/zkQ=]

GoGraph Query Performance

The following performance test cases are taken from the GoGraph's test function suite. The first lot of tests demonstrate some of the implemented functions and the latter tests demonstrate query performance against the Movie database.

Non-Concurrent Query Engine

Git reference: https://github.com/rosshpayne/GoGraph/gql/ast/execute.go

Unlike the RDF Loader the query parse and execution are not concurrent designs. A concurrent design would enable each path in the graph to be executed independently in its own goroutine.

So the non-current design involves firstly executing the root query and for each node returned, walk the graph starting at the node while applying filters as necessary.

Node Cache

Github: https://github.com/rosshpayne/GoGraph/cache

There are no external caches used in testing. While a concurrent non-blocking node cache[12] was developed for GoGraph it was disabled for all tests as the point of the exercise is to quantify the performance of the underlying data stores not the performance of the cache. The query engine does cache nodes after they have been read from the database but unless the query revisits a node this will provide no benefit. This cache is cleared between queries.

A Word About The Data

Two graphs are used for testing. The *Relationship Graph* (https://github.com/rosshpayne/GoGraph/data/person.rdf) was handcrafted specifically for functional testing. It contains only a handful of nodes populated with familiar data and explicitly defined UUIDs to aid in the verification of the test results. *Note: because this graph is aimed at functional testing the query response times are unrealistic due to the limited number of nodes in the graph.*

The *Movie Graph,* on the other hand, is sourced from the internet (https://github.com/rosshpayne/GoGraph/data/million.rdf) and is large enough for performance testing. The internet file is however not in the appropriate format for the concurrent Loader and must be run through the migrate program to regenerate and reorder the RDF tuples it into an acceptable format for the Loader. The migration code is available here: https://github.com/rosshpayne/GoGraph/rdfm

The *Movie Graph* has the following stats.

RDF Triples: 1153863

947 director nodes

70780 actor nodes

283 genre nodes

6356 film nodes

119258 character nodes

119258 performance nodes

7389 film-director edges

119258 film-performance edges

119258 performance-actor edges

119258 performance-film edges

119258 performance-character edges

23679 film-genre edges

[12] Algorithm from the "The Go Programming Language" Alan Donovan, Brian Kernighan

Test 1: Edge Predicate Filter with Boolean expression

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	20
Nodes Displayed (after filtering)	15
Nodes at each depth	{ 3, 7, 10 }
Nodes Displayed at each Depth (after Filtering)	{ 3, 2, 10 }
Reduction in DB requests *	50%
Database requests	1 root request 4 node requests **
Query Elapsed Time (estimated) ***	52 ms
Average Node query time (estimated)	5.2 ms

* as a result of propagating scalar data from child to parent node
** limitation of test data. More realistic graph data would require 10 requests
*** based on 10 nodes queried, none cached.

Comment

This test highlights the use of a boolean expression inside an edge predicate filter.

See comments in red for further explanation.

Test Function

```
func TestUPredFilter4ab(t *testing.T) {                    <= Go Test function

input := `{                                                <= GraphQL statement
  directors(func: eq(count(Siblings), 2) ) {               <= Root expression. Find persons with 2 siblings.
    Age
    Name
    Friends @filter( (le(Age,40) or eq(Name,"Ian Payne")) and ge(Age,62)) { <= Boolean expression Filter
       Age
       Name
       Comment
       Friends   {
         Name
         Age
       }
       Siblings {
         Age
         Name
         Comment
       }
    }
  }
}`

       expectedTouchLvl = []int{3, 2, 10}         <= meta data to validate response. Nodes queried at each depth
       expectedTouchNodes = 15

       stmt := Execute("Relationship", input)     <= Graph to query. Parse and Execute.
       result := stmt.MarshalJSON()               <= Generate JSON output ofresponse
       t.Log(stmt.String())

       validate(t, result)
       Shutdown()

}
```

$ **go test -run=TestUPredFilter4ab -v**

```
Duration:  Parse  87.646126ms  Execute: 27.420353ms      <= Parse and Execute times
```

```
        {
        data: [
                {
                Age : 62,
                Name : "Ross Payne",
                Friends : [
                        {
                        Age: 67,
                        Name: "Ian Payne",
                             Comment: "One of the best cab rides I have Payne seen to date! Anyone know how fast the train was going
around 20 mins in?",
                        Friends : [
                                {
                                Name: "Phil Smith",
                                Age: 36,
                                },
                                {
                                Name: "Ross Payne",
                                Age: 62,
                                },
                                {
                                Name: "Paul Payne",
                                Age: 58,
                                },
                        ],
                        Siblings : [
                                {
                                Age: 58,
                                Name: "Paul Payne",
                                Comment: "A foggy snowy morning lit with Smith sodium lamps is an absolute dream",
                                },
                                {
                                Age: 62,
                                Name: "Ross Payne",
                                 Comment: "Another fun  video. Loved it my Payne Grandmother was from Passau. Dad was over in Germany
but there was something going on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend
everyone.",
                                },
                        ]
                        }
                ]
                },
                {
                Age : 67,
                Name : "Ian Payne",
                Friends : [
                ]
                },
                {
                Age : 58,
                Name : "Paul Payne",
                Friends : [
                        {
                        Age: 67,
                        Name: "Ian Payne",
                             Comment: "One of the best cab rides I have Payne seen to date! Anyone know how fast the train was going
around 20 mins in?",
                        Friends : [
                                {
                                Name: "Phil Smith",
                                Age: 36,
                                },
                                {
                                Name: "Ross Payne",
                                Age: 62,
                                },
                                {
                                Name: "Paul Payne",
                                Age: 58,
                                },
                        ],
                        Siblings : [
                                {
                                Age: 58,
                                Name: "Paul Payne",
                                Comment: "A foggy snowy morning lit with Smith sodium lamps is an absolute dream",
                                },
                                {
                                Age: 62,
                                Name: "Ross Payne",
                                 Comment: "Another fun  video. Loved it my Payne Grandmother was from Passau. Dad was over in Germany
but there was something going on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend
everyone.",
                                },
                        ]
                        }
                ]
                }
        ]
        }

in comparStat  [3 2 10]
--- PASS: TestUPredFilter4ab (2.13s)
PASS
ok      github.com/DynamoGraph/gql      2.170s
```

```
DB:2021/03/27 06:54:42 log.go:89: ================================== SetLogger ==========================================
gqlES: 2021/03/27 06:54:42.181911 Client: 7.9.0                                          <= Version: ElasticSearch client and server
gqlES: 2021/03/27 06:54:42.181938 Server: 7.8.1
gql: 2021/03/27 06:54:42.181951 Startup...
monitor: 2021/03/27 06:54:42.182082 Powering on...                                       <= Start required services: monitor, grmgr
grmgr: 2021/03/27 06:54:42.182099 Powering on...
gql: 2021/03/27 06:54:42.182110 services started
TypesDB: 2021/03/27 06:54:42.182467 db.getGraphId                                        <= Load GraphId, Type data into application
TypesDB: 2021/03/27 06:54:42.248367 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 65.822007ms
TypesDB: 2021/03/27 06:54:42.248506 db.loadTypeShortNames
TypesDB: 2021/03/27 06:54:42.251117 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 2.568258ms
TypesDB: 2021/03/27 06:54:42.254925 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 3.72274ms
gqlDB: 2021/03/27 06:54:42.276772 GSIS:consumed capacity for Query index P_S, {      <= Root query: GSI read
  CapacityUnits: 0.5,                                                                    <= eventually consistent read
  TableName: "DyGraphOD2"
}.  ItemCount 3  Duration: 6.638312ms                                                    <= 3 nodes satisfy root query
DB FetchNode: 2021/03/27 06:54:42.276875  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#  <= Node Query: SortKey beginsWith “A#”
DB: 2021/03/27 06:54:42.282036 FetchNode:consumed capacity for Query  {              <= Query API
  CapacityUnits: 1,                                                                      <= strongly consistent read (configured)
  TableName: “DyGraphOD2"
}. ItemCount 19  Duration: 5.104856ms                                                    <= all node data fetched. 19 items
DB FetchNode: 2021/03/27 06:54:42.282643  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#G#<= read propagated items only
DB: 2021/03/27 06:54:42.288979 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.28986ms                                                     <= 12 items read into execution cache
DB FetchNode: 2021/03/27 06:54:42.289150  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#
grmgr: 2021/03/27 06:54:42.289192 EndCh received for execute. rCnt = 0
DB: 2021/03/27 06:54:42.293341 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 20  Duration: 4.14494ms
grmgr: 2021/03/27 06:54:42.293645 EndCh received for execute. rCnt = -1
DB FetchNode: 2021/03/27 06:54:42.293659  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#  <= read scalar and propagated items
DB: 2021/03/27 06:54:42.297268 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 3.570927ms
gql: 2021/03/27 06:54:42.297492 Duration: Parse  87.646126ms  Execute: 27.420353ms    <= Elapsed times: query parse, query execution
grmgr: 2021/03/27 06:54:42.297502 EndCh received for execute. rCnt = -2
monitor: 2021/03/27 07:00:57.929751 monitor: []interface {}{3, 3, 15, []int{3, 2, 10}, 4, (*monitor.Fetch)(0xc0004c14c0), interface
{}(nil), interface {}(nil)} monitor.Fetch{Fetches:4, CapacityUnits:4, Items:70, Duration:20530844}
gql: 2021/03/27 06:54:44.297627 Shutdown commenced...                                    <= Shutdown services: monitor, grmgr
grmgr: 2021/03/27 06:54:44.297682 Powering down...
monitor: 2021/03/27 06:54:44.297692 Powering down...
gql: 2021/03/27 06:54:44.297701 Shutdown Completed
```

Test 2: Full Text Searching and Root Filter Expression

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	13
Nodes Displayed (after filtering)	12
Nodes at each depth	{ 2, 2, 3, 6 }
Nodes Displayed at each Depth (after Filtering)	{ 1, 2, 3, 6 }
Reduction in DB requests *	46%
Database requests	1 ElasticSearch request 7 Dynamodb requests
Query Elapsed Time	44.5 ms
Average Node query time	2.8 ms (44.5-7.79)/13

* as a result of propagating scalar data from child to parent node

Comment

This test demonstrates uses two data sources to resolve the query. The root query uses the full text search function, *anyofterms,* to query ***ElasticSearch***, which returns with the node UUIDs that satisfy the function. For each UUID GoGraph then searches ***Dynamodb*** to resolve the filter function. Those nodes that satisfy the filter are then passed through the rest of the GraphQL query.

Test Function

```
func TestRootQueryAnyPlusFilter2(t *testing.T) {

        input := `{
   directors(func: anyofterms(Comment,"sodium Germany Chris")) @filter(gt(Age,60)){ <= ElasticSearch query + filter
expression
    Age
    Name
    Comment
    Friends {
         Name
         Age
         Siblings {
                 Name
                  Friends {
                       Name
                       Age
                       Comment
                 }
         }
     }
  }
}`
        expectedTouchLvl = []int{1, 2, 3, 6}
        expectedTouchNodes = 12

        stmt := Execute("Relationship", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)
}
```

$ **go test -run=TestRootQueryAnyPlusFilter2 -v**

```
gql: 2021/03/30 03:29:55.220263 Duration: Parse  76.1568ms  Execute: 44.521291ms
```

monitor: 2021/03/30 03:29:57.220698 monitor: []interface {}{2, 1, 13, []int{2, 2, 3, 6}, 12, []int{1, 2, 3, 6}, 7, (*monitor.Fetch)(0xc0004b1140), interface {}(nil), interface {}(nil)} monitor.Fetch{Fetches:7, CapacityUnits:7, Items:94, Duration:32610743}

```
data: [
                {
                Age : 62,
                Name : "Ross Payne",
                    Comment : "Another fun  video. Loved it my Payne Grandmother was from Passau. Dad was over in Germany but there was something going on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend everyone.",
                Friends : [
                        {
                        Name: "Phil Smith",
                        Age: 36,
                        Siblings : [
                                {
                                Name: "Jenny Jones",
                                Friends : [
                                        {
                                        Name: "Ross Payne",
                                        Age: 62,
                                             Comment: "Another fun  video. Loved it my Payne Grandmother was from Passau. Dad was over in Germany but there was something going on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend everyone.",
                                        },
                                        {
                                        Name: "Paul Payne",
                                        Age: 58,
                                         Comment: "A foggy snowy morning lit with Smith sodium lamps is an absolute dream",
                                        },
                                ]
                                },
                        ]
                        },
                        {
                        Name: "Ian Payne",
                        Age: 67,
                        Siblings : [
                                {
                                Name: "Paul Payne",
                                Friends : [
                                        {
                                        Name: "Ross Payne",
                                        Age: 62,
                                             Comment: "Another fun  video. Loved it my Payne Grandmother was from Passau. Dad was over in Germany but there was something going on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend everyone.",
                                        },
                                        {
                                        Name: "Ian Payne",
                                        Age: 67,
                                         Comment: "One of the best cab rides I have Payne seen to date! Anyone know how fast the train was going around 20 mins in?",
                                        },
                                ]
                                },
                                {
                                Name: "Ross Payne",
                                Friends : [
                                        {
                                        Name: "Phil Smith",
                                        Age: 36,
                                             Comment: "It seems to me the camera's Smith focus is better, clearer, thank you for sharing, I think Austria is a beautiful country I'm not surprised that it produced so many great classical musicians.",
                                        },
                                        {
                                        Name: "Ian Payne",
                                        Age: 67,
                                         Comment: "One of the best cab rides I have Payne seen to date! Anyone know how fast the train was going around 20 mins in?",
                                        },
                                ]
                                },
                        ]
                        }
                ]
                }
        ]
        }
```

Log Output

```
DB:2021/03/30 03:29:55 log.go:89: ========================================= SetLogger
gqlES: 2021/03/30 03:29:55.098713 Client: 7.9.0
gqlES: 2021/03/30 03:29:55.098739 Server: 7.8.1
gql: 2021/03/30 03:29:55.098755 Startup...
monitor: 2021/03/30 03:29:55.098803 Powering on...
grmgr: 2021/03/30 03:29:55.098832 Powering on...
```

```
gql: 2021/03/30 03:29:55.098844 services started
TypesDB: 2021/03/30 03:29:55.099570 db.getGraphId
TypesDB: 2021/03/30 03:29:55.166821 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 67.163807ms
TypesDB: 2021/03/30 03:29:55.166958 db.loadTypeShortNames
TypesDB: 2021/03/30 03:29:55.169869 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 2.871743ms
TypesDB: 2021/03/30 03:29:55.173918 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 3.953619ms
gqlES: 2021/03/30 03:29:55.183507 ES Search duration: 7.79105ms                                  <= ElasticSearch log
entry
DB FetchNode: 2021/03/30 03:29:55.185477  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#
DB: 2021/03/30 03:29:55.192306 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 6.682568ms
grmgr: 2021/03/30 03:29:55.192459 EndCh received for execute. rCnt = 0
DB FetchNode: 2021/03/30 03:29:55.192490  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#
DB: 2021/03/30 03:29:55.198446 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 20  Duration: 5.918953ms
DB FetchNode: 2021/03/30 03:29:55.198616  node: OlTBKXemTNWATwDDt6U5/A== subKey: A#G#
DB: 2021/03/30 03:29:55.203035 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 4.378531ms
DB FetchNode: 2021/03/30 03:29:55.203210  node: 6nG/Cd+dSoyD48CrXQjrLQ== subKey: A#G#
DB: 2021/03/30 03:29:55.206455 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 7  Duration: 3.204773ms
DB FetchNode: 2021/03/30 03:29:55.206625  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#G#
DB: 2021/03/30 03:29:55.211914 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 5.249466ms
DB FetchNode: 2021/03/30 03:29:55.212071  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#G#
DB: 2021/03/30 03:29:55.215904 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.796902ms
DB FetchNode: 2021/03/30 03:29:55.216088  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#G#
DB: 2021/03/30 03:29:55.219507 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.37955ms
grmgr: 2021/03/30 03:29:55.220246 EndCh received for execute. rCnt = -1
gql: 2021/03/30 03:29:55.220263 Duration: Parse  76.1568ms  Execute: 44.521291ms
monitor: 2021/03/30 03:29:57.220698 monitor: []interface {}{2, 1, 13, []int{2, 2, 3, 6}, 12, []int{1, 2, 3, 6}, 7,
(*monitor.Fetch)(0xc0004b1140), interface {}(nil), interface {}(nil)} monitor.Fetch{Fetches:7, CapacityUnits:7,
Items:94, Duration:32610743}
DB: 2021/03/30 03:29:57.228503 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 4,
  TableName: "TestLog"
}. Duration: 7.498688ms
gql: 2021/03/30 03:29:57.228554 Shutdown commenced...
monitor: 2021/03/30 03:29:57.228570 Powering down...
monitor: 2021/03/30 03:29:57.228571 Powering down...
gql: 2021/03/30 03:29:57.228583 Shutdown Completed
```

Test 3: Generate a Graph of Depth 5

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	145
Nodes Displayed (after filtering)	145
Nodes at each depth	{ 3, 7, 30, 32, 73}
Reduction in DB requests *	50%
Database requests	1 Root request 15 Node requests **
Query Elapsed Time	78 ms 340 ms ***
Average Node query time	4.7 ms

* as a result of propagating a child node's scalar data to the parent node
** Due to internal caches in the execution engine only 15 nodes are fetched from Dynamodb. For more realistic data 72 nodes would be fetched.
*** When extrapolated to 72 nodes.

Comment

This query demonstrates generating a graph of depth 5. The root query uses the equality function, *eq*, to find the nodes that have exactly two siblings.

Test Function

```
func TestRootQuery1f(t *testing.T) {

  input := `{
  directors(func: eq(count(Siblings), 2)) {
    Age
    Name
    Friends {
            Age
            Name
            Friends {
                    Name
                    Age
                    Siblings {
                            Name
                            Age
                            Friends {
                                    Name
                                    Age
                                    DOB
                            }
                    }
            }
            Siblings {
                    Name
            }
    }
  }
}`
        expectedTouchLvl = []int{3, 7, 30, 32, 73}
        expectedTouchNodes = 145

        stmt := Execute("Relationship", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)
        Shutdown()
}
```

$ **go test -run=TestRootQuery1f -v**

```
Duration: Parse  85.067525ms  Execute: 78.326803ms

monitor: []interface {}{3, 3, 145, []int{3, 7, 30, 32, 73}, 145, []int{3, 7, 30, 32, 73}, 15, (*monitor.Fetch)(0xc0000f4b20),
interface {}(nil), interface {}(nil)}

      {
        data: [
                {
                Age : 62,
                Name : "Ross Payne",
                Friends : [
                        {
                        Age: 36,
                        Name: "Phil Smith",
                        Friends : [
                                {
                                Name: "Paul Payne",
                                Age: 58,
                                Siblings : [
                                        {
                                        Name: "Ross Payne",
                                        Age: 62,
                                        Friends : [
                                                {
                                                Name: "Phil Smith",
                                                Age: 36,
                                                DOB: "17 June 1976",
                                                },
                                                {
                                                Name: "Ian Payne",
                                                Age: 67,
                                                DOB: "29 Jan 1953",
                                                },
                                        ]
                                        },
                                        {
                                        Name: "Ian Payne",
                                        Age: 67,
                                        Friends : [
                                                {
                                                Name: "Phil Smith",
                                                Age: 36,
                                                DOB: "17 June 1976",
                                                },
                                                {
                                                Name: "Ross Payne",
                                                Age: 62,
                                                DOB: "13 March 1958",
                                                },
                                                {
                                                Name: "Paul Payne",
                                                Age: 58,
                                                DOB: "2 June 1960",
                                                },
                                        ]
                                        },
                                ]


                        . . . .


                                       {
                                        Name: "Ian Payne",
                                        Age: 67,
                                        Friends : [
                                                {
                                                Name: "Phil Smith",
                                                Age: 36,
                                                DOB: "17 June 1976",
                                                },
                                                {
                                                Name: "Ross Payne",
                                                Age: 62,
                                                DOB: "13 March 1958",
                                                },
                                                {
                                                Name: "Paul Payne",
                                                Age: 58,
                                                DOB: "2 June 1960",
                                                },
                                        ]
                                        },
                                ]
                                },
                        ],
                        Siblings : [
                                {
                                Name: "Paul Payne",
                                },
                                {
                                Name: "Ross Payne",
                                },
                        ]
                        }
                ]
                }
        ]
        }
```

Log Output

```
DB:2021/03/31 09:03:18 log.go:89: ================================== SetLogger
```

```
gqlES: 2021/03/31 09:03:18.669729 Client: 7.9.0
gqlES: 2021/03/31 09:03:18.669749 Server: 7.8.1
gql: 2021/03/31 09:03:18.669760 Startup...
monitor: 2021/03/31 09:03:18.669800 Powering on...
monitor: 2021/03/31 09:03:18.669800 Powering on...
gql: 2021/03/31 09:03:18.669826 services started
TypesDB: 2021/03/31 09:03:18.669969 db.getGraphId
TypesDB: 2021/03/31 09:03:18.736810 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 66.755535ms
TypesDB: 2021/03/31 09:03:18.736875 db.loadTypeShortNames
TypesDB: 2021/03/31 09:03:18.740159 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 3.242505ms
TypesDB: 2021/03/31 09:03:18.744088 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 3.842719ms
gqlDB: 2021/03/31 09:03:18.762227 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 3  Duration: 7.148268ms
DB FetchNode: 2021/03/31 09:03:18.762331  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#
DB: 2021/03/31 09:03:18.769233 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 6.848353ms
DB FetchNode: 2021/03/31 09:03:18.769934  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#G#
DB: 2021/03/31 09:03:18.773936 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.954668ms
DB FetchNode: 2021/03/31 09:03:18.774061  node: OlTBKXemTNWATwDDt6U5/A== subKey: A#G#
DB: 2021/03/31 09:03:18.780447 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.341387ms
DB FetchNode: 2021/03/31 09:03:18.780697  node: 6nG/Cd+dSoyD48CrXQjrLQ== subKey: A#G#
DB: 2021/03/31 09:03:18.788195 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 7  Duration: 7.428827ms
DB FetchNode: 2021/03/31 09:03:18.788311  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#G#
DB: 2021/03/31 09:03:18.792966 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 4.605103ms
DB FetchNode: 2021/03/31 09:03:18.793090  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#G#
DB: 2021/03/31 09:03:18.796306 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.174697ms
DB FetchNode: 2021/03/31 09:03:18.796415  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#G#
DB: 2021/03/31 09:03:18.799894 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.437771ms
DB FetchNode: 2021/03/31 09:03:18.800165  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#G#
DB: 2021/03/31 09:03:18.803994 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.783304ms
DB FetchNode: 2021/03/31 09:03:18.804140  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#G#
DB: 2021/03/31 09:03:18.807903 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.725294ms
DB FetchNode: 2021/03/31 09:03:18.808030  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#G#
DB: 2021/03/31 09:03:18.811703 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.633049ms
DB FetchNode: 2021/03/31 09:03:18.811831  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#G#
DB: 2021/03/31 09:03:18.815188 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.317495ms
grmgr: 2021/03/31 09:03:18.815325 EndCh received for execute. rCnt = 0
DB FetchNode: 2021/03/31 09:03:18.815352  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#
DB: 2021/03/31 09:03:18.819317 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 20  Duration: 3.91359ms
DB FetchNode: 2021/03/31 09:03:18.821221  node: OlTBKXemTNWATwDDt6U5/A== subKey: A#G#
DB: 2021/03/31 09:03:18.825456 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 4.183986ms
grmgr: 2021/03/31 09:03:18.825589 EndCh received for execute. rCnt = -1
```

```
DB FetchNode: 2021/03/31 09:03:18.825700  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#
DB: 2021/03/31 09:03:18.829595 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 3.849564ms
DB FetchNode: 2021/03/31 09:03:18.829751  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#G#
DB: 2021/03/31 09:03:18.833218 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.428239ms
gql: 2021/03/31 09:03:18.833344 Duration: Parse  85.067525ms  Execute: 78.326803ms
grmgr: 2021/03/31 09:03:18.833353 EndCh received for execute. rCnt = -2
monitor: 2021/03/31 09:03:20.836150 monitor: []interface {}{3, 3, 145, []int{3, 7, 30, 32, 73}, 145, []int{3, 7,
30, 32, 73}, 15, (*monitor.Fetch)(0xc0000fe260), interface {}(nil), interface {}(nil)} monitor.Fetch{Fetches:15,
CapacityUnits:15, Items:197, Duration:65625327}
DB: 2021/03/31 09:03:20.853483 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 12,
  TableName: "TestLog"
}. Duration: 17.1962ms
gql: 2021/03/31 09:03:20.853519 Shutdown commenced...
grmgr: 2021/03/31 09:03:20.853533 Powering down...
monitor: 2021/03/31 09:03:20.853542 Powering down...
gql: 2021/03/31 09:03:20.853547 Shutdown Completed
```

Test 4: Has Function in Filter

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	5
Nodes Displayed (after filtering)	3
Nodes at each depth	{3, 2}
Nodes at each depthFiltered	{1, 2}
Reduction in DB requests *	40%
Database requests	1 Root request 3 Node requests
Query Elapsed Time	27.5 ms
Average Node query time for 5 nodes **	5.5 ms

* as a result of propagating a child node's scalar data to the parent node
** 3 nodes where fetched from database containing data for 5 nodes because of data propagation from child to parent

Comment

The *has* function checks each node for the existence of a particular attribute. In this example it filters any nodes returned from the root query that do not have an “Address" attribute.

Test Function

```
func TestRootFilterHas1(t *testing.T) {

        input := `{
          me(func: eq(count(Siblings),2)) @filter(has(Address)) { <= find all nodes with 2 siblings that have an address
            Name
            Address
            Age
            Siblings {
                Name
                Age
            }
          }
        }`

        expectedTouchLvl = []int{1, 2}
        expectedTouchNodes = 3

        stmt := Execute("Relationship", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)

}
```

$ **go test -run=TestRootFilterHas1 -v**

Duration: Parse 102.856088ms Execute: 27.534578ms

monitor: []interface {}{3, 1, 5, []int{3, 2}, 3, []int{1, 2}, 3, (*monitor.Fetch)(0xc00030a160), interface {}(nil), interface {}(nil)}

```
{
    data: [
        {
        Name : "Ross Payne",
```

```
        Address : "55 Fredrick St Helensville, QLD, Australia",
        Age : 62,
        Siblings : [
            {
            Name: "Paul Payne",
            Age: 58,
            },
            {
            Name: "Ian Payne",
            Age: 67,
            }
        ]
        }
    ]
}
```

Log Output

```
DB:2021/03/31 03:35:33 log.go:89: ================================== SetLogger
gqlES: 2021/03/31 03:35:33.692714 Client: 7.9.0
gqlES: 2021/03/31 03:35:33.692744 Server: 7.8.1
gql: 2021/03/31 03:35:33.692759 Startup...
monitor: 2021/03/31 03:35:33.692803 Powering on...
grmgr: 2021/03/31 03:35:33.692817 Powering on...
gql: 2021/03/31 03:35:33.692829 services started
TypesDB: 2021/03/31 03:35:33.693018 db.getGraphId
TypesDB: 2021/03/31 03:35:33.772950 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 79.843545ms
TypesDB: 2021/03/31 03:35:33.773099 db.loadTypeShortNames
TypesDB: 2021/03/31 03:35:33.775855 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 2.71016ms
TypesDB: 2021/03/31 03:35:33.785338 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 9.392268ms
gqlDB: 2021/03/31 03:35:33.802703 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 3  Duration: 6.539166ms
DB FetchNode: 2021/03/31 03:35:33.802831  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#
DB: 2021/03/31 03:35:33.811000 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 8.10721ms
grmgr: 2021/03/31 03:35:33.811292 EndCh received for execute. rCnt = 0
DB FetchNode: 2021/03/31 03:35:33.811316  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#
DB: 2021/03/31 03:35:33.815989 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 20  Duration: 4.630716ms
DB FetchNode: 2021/03/31 03:35:33.816146  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#
grmgr: 2021/03/31 03:35:33.816176 EndCh received for execute. rCnt = -1
DB: 2021/03/31 03:35:33.823228 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 7.036968ms
gql: 2021/03/31 03:35:33.823396 Duration: Parse  102.856088ms  Execute: 27.534578ms
grmgr: 2021/03/31 03:35:33.823510 EndCh received for execute. rCnt = -2
monitor:  2021/03/31  03:35:35.823802  monitor:  []interface  {}{3,  1,  5,  []int{3,  2},  3,  []int{1,  2},  3,
(*monitor.Fetch)(0xc00030a160),  interface  {}(nil),  interface  {}(nil)}  monitor.Fetch{Fetches:3,  CapacityUnits:3,
Items:58, Duration:19774894}
DB: 2021/03/31 03:35:35.832098 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 2,
  TableName: "TestLog"
}. Duration: 8.089336ms
```

Test 5: *Has* function in Root Filter and Edge Predicate

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	9
Nodes Displayed (after filtering)	5
Nodes at each depth	{3, 6}
Nodes at each depthFiltered	{3, 2}
Reduction in DB requests *	66%
Database requests	1 Root request 3 Node requests
Query Elapsed Time	24.6 ms
Average Node query time for 9 nodes **	1.9 ms

* as a result of propagating a child node's scalar data to the parent node
** 3 nodes where fetched from database containing data for 9 nodes because of data propagation from child to parent

Comment

Test Function

```
func TestUidPredFilterHasScalar(t *testing.T) {

        input := `{
          me(func: eq(count(Siblings),2)) @filter(has(Friends)) {
                  Name
                 Address
                 Age
                 Siblings @filter(has(Address)) {
                         Name
                         Age
                 }
            }
        }`

        expectedTouchLvl = []int{3, 2}
        expectedTouchNodes = 5

        stmt := Execute("Relationship", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())
        validate(t, result)
}
```

Test Function

```
$ go test -run=TestUidPredFilterHasScalar -v

Duration:  Parse  77.336338ms  Execute: 24.630652ms

monitor: []interface {}{3, 3, 9, []int{3, 6}, 5, []int{3, 2}, 3, (*monitor.Fetch)(0xc000562aa0), interface {}
(nil), interface {}(nil)}

      {
        data: [
                {
                Name : "Ross Payne",
                Address : "67/55 Burkitt St Page, ACT, Australia",
                Age : 62,
```

```
                Siblings : [
                ]
                },
                {
                Name : "Ian Payne",
                 Address : <nil>,
                Age : 67,
                Siblings : [
                        {
                        Name: "Ross Payne",
                        Age: 62,
                        }
                ]
                },
                {
                Name : "Paul Payne",
                 Address : <nil>,
                Age : 58,
                Siblings : [
                        {
                        Name: "Ross Payne",
                        Age: 62,
                        },
                ]
                }
        ]
        }


DB:2021/03/31 04:52:40 log.go:89: ==================================== SetLogger
gqlES: 2021/03/31 04:52:40.651728 Client: 7.9.0
gqlES: 2021/03/31 04:52:40.651748 Server: 7.8.1
gql: 2021/03/31 04:52:40.651759 Startup...
grmgr: 2021/03/31 04:52:40.651810 Powering on...
grmgr: 2021/03/31 04:52:40.651810 Powering on...
gql: 2021/03/31 04:52:40.651853 services started
TypesDB: 2021/03/31 04:52:40.652055 db.getGraphId
TypesDB: 2021/03/31 04:52:40.720110 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 67.963248ms
TypesDB: 2021/03/31 04:52:40.720251 db.loadTypeShortNames
TypesDB: 2021/03/31 04:52:40.723177 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 2.882168ms
TypesDB: 2021/03/31 04:52:40.727393 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 4.111155ms
gqlDB: 2021/03/31 04:52:40.736516 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 3  Duration: 7.086526ms
DB FetchNode: 2021/03/31 04:52:40.736617  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#
DB: 2021/03/31 04:52:40.742904 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 6.240674ms
DB FetchNode: 2021/03/31 04:52:40.743327  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#
grmgr: 2021/03/31 04:52:40.743364 EndCh received for execute. rCnt = 0
DB: 2021/03/31 04:52:40.747538 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 20  Duration: 4.134888ms
grmgr: 2021/03/31 04:52:40.747704 EndCh received for execute. rCnt = -1
DB FetchNode: 2021/03/31 04:52:40.747713  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#
DB: 2021/03/31 04:52:40.753837 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 19  Duration: 6.082396ms
gql: 2021/03/31 04:52:40.753992 Duration: Parse  77.336338ms  Execute: 24.630652ms
grmgr: 2021/03/31 04:52:40.754002 EndCh received for execute. rCnt = -2
monitor:  2021/03/31  04:52:42.754314  monitor:  []interface  {}{3,  3,  9,  []int{3,  6},  5,  []int{3,  2},  3,
(*monitor.Fetch)(0xc000562aa0),  interface  {}(nil),  interface  {}(nil)}  monitor.Fetch{Fetches:3,  CapacityUnits:3,
Items:58,  Duration:16958}
DB: 2021/03/31 04:52:42.762577 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 2,
  TableName: "TestLog"
}. Duration: 8.036407ms
```

Test 6: *AnyOfTerms* function in Root Filter

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	3
Nodes Displayed (after filtering)	2
Nodes at each depth	{ 3 }
Nodes at each depthFiltered	{ 2 }
Reduction in DB requests *	0%
Database requests	1 Root request 3 Node requests

* as a result of propagating a child node's scalar data to the parent node

Comment

Demonstrates the use of the *anyofterms* in a root filter. While the root query is resolved in Dynamodb the filter function is implemented in GoGraph.

Test Function

```
func TestRootFilteranyofterms1(t *testing.T) {

         input := `{
           me(func: eq(count(Siblings),2)) @filter( anyofterms(Comment,"sodium Germany Chris") ) {
                           Name
                 Comment
             }
           }`

         expectedTouchLvl = []int{2}
         expectedTouchNodes = 2

         stmt := Execute("Relationship", input)
         result := stmt.MarshalJSON()
         t.Log(stmt.String())

         validate(t, result)

}
```

$ **go test -run=TestRootFilteranyofterms1 -v**

Duration: Parse 75.978328ms Execute: 21.894101ms

monitor: []interface {}{3, 2, 3, []int{3}, 2, []int{2}, 3, (*monitor.Fetch)(0xc00020c2a0), interface {}(nil), interface {}(nil)}
in comparStat [2]


```
     {
     data: [
          {
          Name : "Ross Payne",
            Comment : "Another fun  video. Loved it my Payne Grandmother was from Passau. Dad was over in Germany but there was something going
on over there at the time we won't discuss right now. Thanks for posting it. Have a great weekend everyone.",
          },
          {
          Name : "Paul Payne",
          Comment : "A foggy snowy morning lit with Smith sodium lamps is an absolute dream",
          }
     ]
     }
```

```
DB:2021/03/31 05:43:33 log.go:89: ================================== SetLogger
gqlES: 2021/03/31 05:43:33.319823 Client: 7.9.0
gqlES: 2021/03/31 05:43:33.319843 Server: 7.8.1
gql: 2021/03/31 05:43:33.319851 Startup...
monitor: 2021/03/31 05:43:33.319889 Powering on...
monitor: 2021/03/31 05:43:33.319889 Powering on...
gql: 2021/03/31 05:43:33.319918 services started
TypesDB: 2021/03/31 05:43:33.320116 db.getGraphId
TypesDB: 2021/03/31 05:43:33.387590 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 67.398378ms
TypesDB: 2021/03/31 05:43:33.387852 db.loadTypeShortNames
TypesDB: 2021/03/31 05:43:33.391090 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 3.195452ms
TypesDB: 2021/03/31 05:43:33.394467 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 3.282593ms
gqlDB: 2021/03/31 05:43:33.402388 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 3  Duration: 6.281371ms
DB FetchNode: 2021/03/31 05:43:33.402457  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#A
DB: 2021/03/31 05:43:33.408876 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 7  Duration: 6.384021ms
DB FetchNode: 2021/03/31 05:43:33.409059  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#A
grmgr: 2021/03/31 05:43:33.409302 EndCh received for execute. rCnt = 0
DB: 2021/03/31 05:43:33.413977 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 8  Duration: 4.858527ms
DB FetchNode: 2021/03/31 05:43:33.414086  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#A
grmgr: 2021/03/31 05:43:33.414161 EndCh received for execute. rCnt = -1
DB: 2021/03/31 05:43:33.417852 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 7  Duration: 3.72264ms
gql: 2021/03/31 05:43:33.417945 Duration: Parse  75.978328ms  Execute: 21.894101ms
grmgr: 2021/03/31 05:43:33.417954 EndCh received for execute. rCnt = -2
monitor: 2021/03/31 05:43:35.418815 monitor: []interface {}{3, 2, 3, []int{3}, 2, []int{2}, 3, (*monitor.Fetch)
(0xc00020c2a0), interface {}(nil), interface {}(nil)} monitor.Fetch{Fetches:3, CapacityUnits:3, Items:22,
Duration:14965188}
DB: 2021/03/31 05:43:35.427508 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 2,
  TableName: "TestLog"
}. Duration: 8.282496ms
```

Test 8: *AllOfTerms* function in Root Filter

Query Metrics

Metric	Value
Graph	Relationship
Nodes Queried	3
Nodes Displayed (after filtering)	1
Nodes at each depth	{ 3 }
Nodes at each depthFiltered	{ 1 }
Reduction in DB requests *	0%
Database requests	1 Root request 3 Node requests

* as a result of propagating a child node's scalar data to the parent node

Comment

Demonstrates the use of the *allofterms* in a root filter. While the root query is resolved in *Dynamodb* the filter function is implemented in GoGraph.

Test Function

```
func TestRootFilterallofterms1c(t *testing.T) {

input := `{
  me(func: eq(count(Siblings),2)) @filter( allofterms(Comment,"sodium Germany Chris") or eq(Name,"Ian Payne")) {
            Name
            }
          }`

         expectedTouchLvl = []int{1}
         expectedTouchNodes = 1

         stmt := Execute("Relationship", input)
         result := stmt.MarshalJSON()
         t.Log(stmt.String())

         validate(t, result)

}
```

$ **go test -run=TestRootFilterallofterms1c -v**

```
Duration:  Parse  96.683441ms  Execute: 24.462418ms

monitor: []interface {}{3, 1, 3, []int{3}, 1, []int{1}, 3, (*monitor.Fetch)(0xc00002c540), interface {}(nil),
interface {}(nil)}

        {
        data: [
                {
                Name : "Ian Payne",
                }
        ]
        }
```

Log Output

```
DB:2021/04/01 04:20:03 log.go:89: ==================================== SetLogger
gqlES: 2021/04/01 04:20:03.427602 Client: 7.9.0
gqlES: 2021/04/01 04:20:03.427632 Server: 7.8.1
gql: 2021/04/01 04:20:03.427646 Startup...
monitor: 2021/04/01 04:20:03.427688 Powering on...
grmgr: 2021/04/01 04:20:03.427703 Powering on...
gql: 2021/04/01 04:20:03.427714 services started
TypesDB: 2021/04/01 04:20:03.428375 db.getGraphId
```

```
TypesDB: 2021/04/01 04:20:03.504709 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 76.1864ms
TypesDB: 2021/04/01 04:20:03.504779 db.loadTypeShortNames
TypesDB: 2021/04/01 04:20:03.508742 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 3.913565ms
TypesDB: 2021/04/01 04:20:03.514310 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 10 Duration: 5.057964ms
gqlDB: 2021/04/01 04:20:03.530796 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 3  Duration: 5.830418ms
DB FetchNode: 2021/04/01 04:20:03.530891  node: TsI2Qay8T2SxnOcuI3pPeg== subKey: A#A
DB: 2021/04/01 04:20:03.535560 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 7  Duration: 4.61992ms
DB FetchNode: 2021/04/01 04:20:03.536514  node: 66PNdV1TSKOpDRlO71+Aow== subKey: A#A
DB: 2021/04/01 04:20:03.543867 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 8  Duration: 7.227218ms
DB FetchNode: 2021/04/01 04:20:03.544370  node: 6KIuWuyTRKSgDqwYAghl7A== subKey: A#A
DB: 2021/04/01 04:20:03.549178 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 7  Duration: 4.748581ms
gql: 2021/04/01 04:20:03.549368 Duration: Parse  96.683441ms  Execute: 24.462418ms
monitor: 2021/04/01 04:20:05.549638 monitor: []interface {}{3, 1, 3, []int{3}, 1, []int{1}, 3, (*monitor.Fetch)
(0xc000292ba0), interface {}(nil), interface {}(nil)} monitor.Fetch{Fetches:3, CapacityUnits:3, Items:22,
Duration:16595719}
DB: 2021/04/01 04:20:05.561143 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 2,
  TableName: "TestLog"
}. Duration: 11.347616ms
```

Test 9: Equality Expression in Root Query

Query Metrics

Metric	Value
Graph	Movies
Nodes Queried	8
Nodes at each depth	{1, 7}
Database requests	1 Root request 1 Node requests
Reduction in DB requests *	87.5%
Query Elapsed Time	12.5 ms
DB Average Request time for 1 node	5.2 ms
Average query time for 8 nodes	0.74 ms

* as a result of propagating a child node's scalar data to the parent node

Comment

The test demonstrates the use of the equality function *eq* in the root query.

It is also worthwhile mentioning the affect of data duplication (propagating child scalar data to parent) in this test. One node is returned from the root query but embedded in the response is the scalar data of the 7 child “genre” nodes, for a reduction in database query calls of 87.5%.

Test Function

```
func TestMovieEq(t *testing.T) {

input := `{
  me(func:eq(title, "Poison")) {
                               title
                               film.genre {
                                      name
                               }
  }
}`

        expectedTouchLvl = []int{1, 7}
        expectedTouchNodes = 8

        stmt := Execute("Movies", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)

}
```

$ **go test -run=TestMovieEq -v**

```
Duration:  Parse  85.993172ms  Execute: 17.19818ms
monitor: []interface {}{1, 1, 8, []int{1, 7}, 8, []int{1, 7}, 1, (*monitor.Fetch)(0xc0003364c0),
interface {}(nil), interface {}(nil)}

        {
       data: [
               {
               title : "Poison",
               film.genre : [
                       {
                       name: "Satire",
                       },
                       {
                       name: "LGBT",
                       },
```

```
                        {
                        name: "Indie film",
                        },
                        {
                        name: "Science Fiction",
                        },
                        {
                        name: "Drama",
                        },
                        {
                        name: "Experimental film",
                        },
                        {
                        name: "Horror",
                        }
                ]
                }
        ]
        }
```

Log Output

```
DB:2021/04/01 04:31:04 log.go:89: ==================================== SetLogger
gqlES: 2021/04/01 04:31:04.434237 Client: 7.9.0
gqlES: 2021/04/01 04:31:04.434267 Server: 7.8.1
gql: 2021/04/01 04:31:04.434281 Startup...
monitor: 2021/04/01 04:31:04.434330 Powering on...
grmgr: 2021/04/01 04:31:04.434344 Powering on...
gql: 2021/04/01 04:31:04.434354 services started
TypesDB: 2021/04/01 04:31:04.434971 db.getGraphId
TypesDB: 2021/04/01 04:31:04.504133 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 69.074044ms
TypesDB: 2021/04/01 04:31:04.504277 db.loadTypeShortNames
TypesDB: 2021/04/01 04:31:04.508164 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 5 Duration: 3.832432ms
TypesDB: 2021/04/01 04:31:04.512256 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 16 Duration: 3.508138ms
gqlDB: 2021/04/01 04:31:04.521130 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 1  Duration: 6.969338ms
DB FetchNode: 2021/04/01 04:31:04.521205  node: sHoW3PRSQUuYvKC+FwcAHg== subKey: A#
DB: 2021/04/01 04:31:04.526427 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 5.165045ms
gql: 2021/04/01 04:31:04.526621 Duration: Parse  79.214802ms  Execute: 12.498567ms
monitor:  2021/04/01  04:31:06.527012  monitor:  []interface  {}{1,  1,  8,  []int{1,  7},  8,  []int{1,  7},  1,
(*monitor.Fetch)(0xc0004e73e0),  interface  {}(nil),  interface  {}(nil)}  monitor.Fetch{Fetches:1,  CapacityUnits:1,
Items:15,  Duration:5165045}
DB: 2021/04/01 04:31:06.535189 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 2,
  TableName: "TestLog"
}. Duration: 7.902135ms
```

Test 10: AnyOfTerms Function used in Edge Predicate Filter

Query Metrics

Metric	Value
Graph	Movies
Nodes Queried	31 (27 filtered out)
Nodes at each depth	{1, 30}
DB Requests	1 Root request 1 Node requests
Reduction in DB requests *	96.7%
Query Elapsed Time	15.9 ms
DB Request time for 1 node	8 ms
Average Node query time for 31 nodes	0.26 ms

* as a result of propagating a child node's scalar data to the parent node

Comment

Function *AnyOfTerms* is implemented in GoGraph not *Dynamodb*. In this example, all film titles are checked for the words “War” or “Minority”.

The test is also another example of the power of duplicating child scale data in its parent node. In this example 1 node is fetched from the database but embedded in the response is the scalar data of 30 child nodes, reducing database calls query calls by 96.7%.

Test Function

```
Query: show all films by Steven Spielberg with “War” or “Minority” in the title

func TestMovie1b(t *testing.T) {

        input := `{
  me(func: eq(name, "Steven Spielberg")) @filter(has(director.film)) {
    name
    director.film @filter(anyofterms(title,"War Minority") {
      title
    }
  }
}`

        expectedTouchLvl = []int{1, 3}
        expectedTouchNodes = 4

        stmt := Execute("Movies", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)

}
```

$ go test -run=TestMovie1b -v

```
Duration:  Parse  90.957263ms  Execute: 15.889374ms

monitor: []interface {}{1, 1, 31, []int{1, 30}, 4, []int{1, 3}, 1, (*monitor.Fetch)(0xc000205d40), interface {}(nil), interface {}
(nil)}

       {
       data: [
               {
               name : "Steven Spielberg",
               director.film : [
                       {
                       title: "War of the Worlds",
```

```
                    },
                    {
                    title: "Minority Report",
                    },
                    {
                    title: "War Horse",
                    },
                ]
                }
        ]
        }
```

Log Output

```
DB:2021/04/01 04:49:42 log.go:89: ================================== SetLogger
gqlES: 2021/04/01 04:49:42.748304 Client: 7.9.0
gqlES: 2021/04/01 04:49:42.748337 Server: 7.8.1
gql: 2021/04/01 04:49:42.748356 Startup...
grmgr: 2021/04/01 04:49:42.748400 Powering on...
monitor: 2021/04/01 04:49:42.748412 Powering on...
gql: 2021/04/01 04:49:42.748427 services started
TypesDB: 2021/04/01 04:49:42.749136 db.getGraphId
TypesDB: 2021/04/01 04:49:42.819382 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 70.151441ms
TypesDB: 2021/04/01 04:49:42.819441 db.loadTypeShortNames
TypesDB: 2021/04/01 04:49:42.824586 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 5 Duration: 5.097431ms
TypesDB: 2021/04/01 04:49:42.829502 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 16 Duration: 4.754526ms
gqlDB: 2021/04/01 04:49:42.847841 GSIS:consumed capacity for Query index P_S, {  <= index read for root query
  CapacityUnits: 0.5,                                                          <= <4KB read (eventual consistency)
  TableName: "DyGraphOD2"
}.  ItemCount 1  Duration: 7.73318ms                                                 <= 1 item fetched
DB FetchNode: 2021/04/01 04:49:42.847917  node: Yw9EZVUiS6K4yCd+41frbQ== subKey: A#
DB: 2021/04/01 04:49:42.855442 FetchNode:consumed capacity for Query  {              <= node query
  CapacityUnits: 1,                                                            <= <8KB read (eventual consistency)
  TableName: "DyGraphOD2"
}. ItemCount 13  Duration: 7.481137ms                                                <= 13 items fetched
gql: 2021/04/01 04:49:42.856061 Duration: Parse  90.957263ms  Execute: 15.889374ms
monitor:  2021/04/01  04:49:44.856554  monitor:  []interface  {}{1,  1,  31,  []int{1,  30},  4,  []int{1,  3},  1,
(*monitor.Fetch)(0xc000205d40),  interface  {}(nil),  interface  {}(nil)}  monitor.Fetch{Fetches:1,  CapacityUnits:1,
Items:13, Duration:7481137}
DB: 2021/04/01 04:49:44.867515 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 2,
  TableName: "TestLog"
}. Duration: 10.792321ms
```

Test 11: Propagating Child Scalar Data to Parent

Query Metrics

Metric	Value
Graph	Movies
Nodes Queried	90
Nodes at each depth	{ 6, 84 }
Reduction in DB requests *	93%
Database requests	1 Root request 6 Node requests
Query Elapsed Time	64.6 ms
Average Node request time for 6 nodes	7.9 ms
Average Node request time for 90 nodes	0.55 ms

* as a result of propagating a child node's scalar data to the parent node

Comment

This test demonstrates the power of duplicating data (propagating scalar data from child node to parent node). Retrieve 6 nodes from *Dynamodb* and 84 child nodes will be included in the result because of data duplication in the parent nodes. So, query 6 and get 90 nodes back which represents a 93% reduction in database requests.

Test Function

```
Query: show the directors name and genre names of any Movie  with 13 genre associated with it.

func TestMovie1e(t *testing.T) {

        input := `{
  me(func: eq(count(film.genre), 13)) {
    title
    film.director {
        name
    }
    film.genre {
        name
    }
  }
}`

        expectedTouchLvl = []int{6, 84}
        expectedTouchNodes = 90

        stmt := Execute("Movies", input)
        t.Log(stmt.String())
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)

}
```

$ go test -run=TestMovie1e -v

```
Duration:  Parse  80.3016ms  Execute: 64.568085ms

monitor: []interface {}{6, 6, 90, []int{6, 84}, 90, []int{6, 84}, 6, (*monitor.Fetch)(0xc00025a000), interface {}(nil), interface {}
(nil)}

        {
        data: [
                {
                title : "South Park: Bigger, Longer & Uncut",
                film.director : [
                        {
                        name: "Trey Parker",
```

```
        }
film.genre : [
        {
        name: "War film",
        },
        {
        name: "Black comedy",
        },
        {
        name: "Musical comedy",
        },
        {
        name: "Animation",
        },
        {
        name: "Animated Musical",
        },
        {
        name: "Parody",
        },
        {
        name: "Gross out",
        },
        {
        name: "Satire",
        },
        {
        name: "Absurdism",
        },
        {
        name: "Comedy",
        },
        {
        name: "Political cinema",
        },
        {
        name: "Political satire",
        },
        {
        name: "Backstage Musical",
        }
]
},
{
title : "Stay Tuned",
film.director : [
        {
        name: "Chuck Jones",
        }
film.genre : [
        {
        name: "Black comedy",
        },
        {
        name: "Parody",
        },
        {
        name: "Musical comedy",
        },
        {
        name: "Satire",
        },
        {
        name: "Horror comedy",
        },
        {
        name: "Fantasy",
        },
        {
        name: "Family",
        },
        {
        name: "Adventure Film",
        },
        {
        name: "Science Fiction",
        },
        {
        name: "Horror",
        },
        {
        name: "Thriller",
        },
        {
        name: "Comedy",
        },
        {
        name: "Backstage Musical",
        }
]
},
{
title : "Even Cowgirls Get the Blues",
film.director : [
        {
        name: "Gus Van Sant",
        }
film.genre : [
        {
        name: "Feminist Film",
        },
        {
        name: "Romantic comedy",
        },
```

```
                    {
                    name: "Comedy Western",
                    },
                    {
                    name: "Absurdism",
                    },
                    {
                    name: "Comedy-drama",
                    },
                    {
                    name: "Fantasy",
                    },
                   . . . .
```

Log Output

```
DB:2021/04/01 05:22:33 log.go:89: ==================================== SetLogger
gqlES: 2021/04/01 05:22:33.463289 Client: 7.9.0
gqlES: 2021/04/01 05:22:33.463310 Server: 7.8.1
gql: 2021/04/01 05:22:33.463320 Startup...
monitor: 2021/04/01 05:22:33.463359 Powering on...
monitor: 2021/04/01 05:22:33.463362 Powering on...
gql: 2021/04/01 05:22:33.463389 services started
TypesDB: 2021/04/01 05:22:33.463571 db.getGraphId
TypesDB: 2021/04/01 05:22:33.532893 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 69.197451ms
TypesDB: 2021/04/01 05:22:33.532958 db.loadTypeShortNames
TypesDB: 2021/04/01 05:22:33.536267 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 5 Duration: 3.272282ms
TypesDB: 2021/04/01 05:22:33.542444 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 16 Duration: 6.056022ms
gqlDB: 2021/04/01 05:22:33.558967 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 6  Duration: 15.073845ms
DB FetchNode: 2021/04/01 05:22:33.559058  node: S40/DoLkRWuqjvPnrlFNWw== subKey: A#
DB: 2021/04/01 05:22:33.565499 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 6.394954ms
DB FetchNode: 2021/04/01 05:22:33.565718  node: u5vWespTQSigEJqIBwBUFA== subKey: A#
DB: 2021/04/01 05:22:33.575954 FetchNode:consumed capacity for Query  {
  CapacityUnits: 3,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 10.197994ms
DB FetchNode: 2021/04/01 05:22:33.576371  node: 98wbbFQmTcqXUzxOwvOVzg== subKey: A#
DB: 2021/04/01 05:22:33.588027 FetchNode:consumed capacity for Query  {
  CapacityUnits: 3,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 11.60112ms
DB FetchNode: 2021/04/01 05:22:33.588310  node: djTkRRQUS5Gfuz0vdSLXFQ== subKey: A#
DB: 2021/04/01 05:22:33.593769 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 5.415528ms
DB FetchNode: 2021/04/01 05:22:33.594029  node: MB70yljLTjOsh+tfQIxoNQ== subKey: A#
DB: 2021/04/01 05:22:33.598939 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 4.868996ms
DB FetchNode: 2021/04/01 05:22:33.599091  node: axtdNL3WT3mshCK8EUAxzg== subKey: A#
DB: 2021/04/01 05:22:33.608154 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
}. ItemCount 15  Duration: 9.028703ms
gql: 2021/04/01 05:22:33.608421 Duration: Parse  80.3016ms  Execute: 64.568085ms
monitor:  2021/04/01  05:22:35.611330  monitor:  []interface  {}{6,  6,  90,  []int{6,  84},  90,  []int{6,  84},  6,
(*monitor.Fetch)(0xc00025a000),  interface  {}(nil),  interface  {}(nil)}  monitor.Fetch{Fetches:6,  CapacityUnits:12,
Items:90,  Duration:47507295}
DB: 2021/04/01 05:22:35.619636 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 5,
  TableName: "TestLog"
}. Duration: 8.156251ms
```

Test 12: Propagate scalar data to grandparent for 1:1 relationship

Query Metrics

Metric	Value
Graph	Movies
Nodes Touched	1166
Nodes at each depth	{ 1, 15, 15, 391, 744 }
Database requests	1 Root request 14 Node requests
Reduction in DB requests *	99.1%
Query Elapsed Time	122 ms
Average DB Request time	6.6 ms
Average Node Request time	0.10 ms
Read Capacity Units	23

* as a result of propagating scalar data from child to grandparent node
**

Comment

Examine the GraphQL query below and you will see the resulting graph output will have a depth of 5. The three paths in the output graph are shown below:

Depth 4: person->performance->film->director(name)
Depth 5: person->performance->film->performance->person(name)
Depth 5: person->performance->film->performance->character(name)

The following edges, *performance->film*, *performance->actor* and *performance->character* all have a 1:1 relationship (see type definition for Movie). For a 1:1 relationship GoGraph can propagate the child scale data all the way up to the grandparent node which in this case would be *film.* This means the execution algorithm does not need to follow the path from *film* nodes to its leaf nodes as all the intervening data has been duplicated in the *film* node. This will naturally benefit the response time as 391+744+14 nodes have had their scalar data duplicated into 14 film nodes and do not need to be queried. The result is that all 1166 nodes can be queried in around 122ms because 99.1% of nodes do not need to be queried from the database.

To help quantify the advantage of propagating to the grandparent in the case of 1:1 relationships, the next example performs the same query but has disabled child to grandparent propagation for 1:1 and instead implements the normal child to parent node propagation. In this case the query response took 1.61s which is over **13 times slower**. The other significant factor is a huge reduction in consumed *Dynamodb* read-capacity-units, 23 compared to 359 (see monitor output at end of log output in bold red), that is a **93.6% reduction in AWS costs**, when child-grandparent propagation is enabled for 1:1 relationships.

Test Function

```
Query: Find all the films Peter Sellers performed in along with the director's name and list all the actors and
their character they played in each film

func TestMoviePS3a(t *testing.T) {

        input := `{
  me(func: eq(name,"Peter Sellers") ) {
    name
    actor.performance {
        performance.film  {
                title
                film.director {
                        name
```

```
                }
                film.performance {
                        performance.actor {
                                name
                        }
                        performance.character {
                                name
                        }
                }
        }
   }
 }
}`

        expectedTouchLvl = []int{1, 15, 15, 391, 744}
        expectedTouchNodes = 1166

        stmt := Execute("Movies", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)

}
```

$ **go test -run=TestMoviePS3a -v**

```
       {
        data: [
                {
                name : "Peter Sellers",
                actor.performance : [
                        {
                        performance.film : [
                                {
                                title: "Where Does It Hurt?",
                                film.director : [
                                        {
                                        name: "Rod Amateau",
                                        },
                                ],
                                film.performance : [
                                        {
                                        performance.actor : [
                                                {
                                                name: "Harold Gould",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Dr. Zerny",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Rick Lenz",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Lester Hammond",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Keith Allison",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Hinkley",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Hope Summers",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Nurse Throttle",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Jo Ann Pflug",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Alice Gilligan",
                                                },
```

```
                        ]
                        },
                        {
                        performance.actor : [
                                {
                                name: "Paul Lambert",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Dr. Pinikhes",
                                },
                        ]
                        },
                        {
                        performance.actor : [
                                {
                                name: "Peter Sellers",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Dr. Albert T. Hopfnagel",
                                },
                        ]
                        },
                        {
                        performance.actor : [
                                {
                                name: "Pat Morita",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Nishimoto",
                                },
                        ]
                        },
                        {
                        performance.actor : [
                                {
                                name: "Eve Bruce",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Lamarr",
                                },
                        ]
                        },
                        {
                        performance.actor : [
                                {
                                name: "Norman Alden",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Katzen",
                                },
                        ]
                        },
                ]
                },
        ]
        },
        {
        performance.film : [
                {
                title: "Revenge of the Pink Panther",
                film.director : [
                        {
                        name: "Blake Edwards",
                        },
                ],
                film.performance : [
                        {
                        performance.actor : [
                                {
                                name: "Keen Jing",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Assistant Manager",
                                },
                        ]
                        },
                        {
                        performance.actor : [
                                {
                                name: "Tony Beckley",
                                },
                        ],
                        performance.character : [
                                {
                                name: "Guy Algo",
                                },
                        ]
                        },
           . . . . . .
        ]
        },
        {
```

```
performance.film : [
        {
        title: "Never Let Go",
        film.director : [
                {
                name: "John Guillermin",
                },
        ],
        film.performance : [
                {
                performance.actor : [
                        {
                        name: "Noel Willman",
                        },
                ],
                performance.character : [
                        {
                        name: "Inspector Thomas",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "John Bailey",
                        },
                ],
                performance.character : [
                        {
                        name: "Mckinnon",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "Mignon O'Doherty",
                        },
                ],
                performance.character : [
                        {
                        name: "Manageress",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "Charles Houston",
                        },
                ],
                performance.character : [
                        {
                        name: "Cyril Spink",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "Mervyn Johns",
                        },
                ],
                performance.character : [
                        {
                        name: "Alfie Barnes",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "Adam Faith",
                        },
                ],
                performance.character : [
                        {
                        name: "Tommy Towers",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "Nigel Stock",
                        },
                ],
                performance.character : [
                        {
                        name: "Regan",
                        },
                ]
                },
                {
                performance.actor : [
                        {
                        name: "Elizabeth Sellars",
                        },
                ],
                performance.character : [
                        {
                        name: "Anne Cummings",
                        },
                ]
```

```
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "John Le Mesurier",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Pennington",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Peter Jones",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Alec Berger",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Cyril Shaps",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Cypriot",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "David Lodge",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Cliff",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Richard Todd",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "John Cummings",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Peter Sellers",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Lionel Meadows",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "Carol White",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Jackie",
                                                },
                                        ]
                                        },
                                        {
                                        performance.actor : [
                                                {
                                                name: "John Dunbar",
                                                },
                                        ],
                                        performance.character : [
                                                {
                                                name: "Station Sergeant",
                                                },
                                        ]
                                        },
                                ]
                                },
                        ]
                        }
                ]
                }
        ]
```

```
    }
```

Log Output

```
DB:2021/04/01 22:00:47 log.go:89: ================================== SetLogger
gqlES: 2021/04/01 22:00:47.490432 Client: 7.9.0
gqlES: 2021/04/01 22:00:47.490461 Server: 7.8.1
gql: 2021/04/01 22:00:47.490475 Startup...
grmgr: 2021/04/01 22:00:47.490578 Powering on...
monitor: 2021/04/01 22:00:47.490593 Powering on...
gql: 2021/04/01 22:00:47.490606 services started
TypesDB: 2021/04/01 22:00:47.491164 db.getGraphId
TypesDB: 2021/04/01 22:00:47.562158 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 70.870534ms
TypesDB: 2021/04/01 22:00:47.562230 db.loadTypeShortNames
TypesDB: 2021/04/01 22:00:47.565497 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 5 Duration: 3.229603ms
TypesDB: 2021/04/01 22:00:47.569504 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 16 Duration: 3.877985ms
gqlDB: 2021/04/01 22:00:47.587483 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD2"
}.  ItemCount 1  Duration: 7.043006ms
DB FetchNode: 2021/04/01 22:00:47.587556  node: eneJBCenT1qrdnv4X//RLw== subKey: A#
DB: 2021/04/01 22:00:47.595070 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 11  Duration: 6.982168ms
DB FetchNode: 2021/04/01 22:00:47.595294  node: FZBl1d6RSuGcDFlf4khLZQ== subKey: A#G#
DB: 2021/04/01 22:00:47.602002 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.357901ms
DB FetchNode: 2021/04/01 22:00:47.602358  node: k/4DFiGiQl21KIN++n5ApQ== subKey: A#G#
DB: 2021/04/01 22:00:47.612529 FetchNode:consumed capacity for Query  {
  CapacityUnits: 4,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 10.119915ms
DB FetchNode: 2021/04/01 22:00:47.616133  node: ymanEWWqSFSml9xvZK9tsA== subKey: A#G#
DB: 2021/04/01 22:00:47.623994 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,                                                          <= less than 8KB read
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 7.79995ms
DB FetchNode: 2021/04/01 22:00:47.624582  node: 4+rG5lQbQKuoSF0sMLeQhQ== subKey: A#G#
DB: 2021/04/01 22:00:47.630979 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,                                                          <= less than 4KB read
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.353675ms                                       <= 12 items read in 6.35 ms
DB FetchNode: 2021/04/01 22:00:47.631389  node: Nid9drj0Q/eCEHzR3aT1Yg== subKey: A#G#
DB: 2021/04/01 22:00:47.638431 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.993284ms
DB FetchNode: 2021/04/01 22:00:47.643496  node: 52APy4uhSpCp8DhmNt8jaA== subKey: A#G#
DB: 2021/04/01 22:00:47.652697 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 9.147216ms
DB FetchNode: 2021/04/01 22:00:47.656289  node: yDbaVHDsSgmDjvUaBGasCA== subKey: A#G#
DB: 2021/04/01 22:00:47.663093 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.721617ms
DB FetchNode: 2021/04/01 22:00:47.665111  node: ERxPeI++RNO1JXUksF0flQ== subKey: A#G#
DB: 2021/04/01 22:00:47.669605 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 4.441593ms
DB FetchNode: 2021/04/01 22:00:47.670316  node: VUgeuT2YQbataNKytyUzww== subKey: A#G#
DB: 2021/04/01 22:00:47.677663 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 7.291921ms
DB FetchNode: 2021/04/01 22:00:47.678734  node: eZTcsNshQ7+/mBTKi7RH/w== subKey: A#G#
DB: 2021/04/01 22:00:47.685135 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 6.353274ms
DB FetchNode: 2021/04/01 22:00:47.685742  node: uzlXBfnlSuWrhpE/ynkC5Q== subKey: A#G#
DB: 2021/04/01 22:00:47.691773 FetchNode:consumed capacity for Query  {
  CapacityUnits: 2,
  TableName: "DyGraphOD2"
```

```
}. ItemCount 12  Duration: 5.981037ms
DB FetchNode: 2021/04/01 22:00:47.692367  node: ocpPHhtCTZ2AMy/XBPe1tw== subKey: A#G#
DB: 2021/04/01 22:00:47.696371 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 3.961547ms
DB FetchNode: 2021/04/01 22:00:47.696814  node: CS65MuW+RgqcOnqKWbWLfw== subKey: A#G#
DB: 2021/04/01 22:00:47.701538 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD2"
}. ItemCount 12  Duration: 4.686253ms
gql: 2021/04/01 22:00:47.702873 Duration: Parse  89.276114ms  Execute: 122.473852ms
monitor: 2021/04/01 22:00:50.372524 monitor: []interface {}{1, 1, 1166, []int{1, 15, 15, 391, 744}, 1166, []int{1,
15,  15,  391,  744},  14,  (*monitor.Fetch)(0xc0004cc400),  interface  {}(nil),  interface  {}(nil)}
monitor.Fetch{Fetches:14, CapacityUnits:23, Items:167, Duration:93191351}
DB: 2021/04/01 22:00:50.403159 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 102,
  TableName: "TestLog"
}. Duration: 30.456313m
```

Test 13: Disabled child-to-grandparent propagation for 1:1 relationship.

Query Metrics

Metric	Value
Graph	Movies
Nodes Queried	1166
Nodes at each depth	{ 1, 15, 15, 391, 744 }
Database requests	1 Root request 359 Node requests
Reduction in DB requests *	65%
Query Elapsed Time	1.61 s
Average DB Request time	4.48 ms
Average Node Request time	1.38 ms
Read Capacity Units	359

* as a result of propagating scalar data from child to parent node

Comment

This example is used to quantity the performance advantage of child to grandparent propagation for 1:1 edge relationships when it is disabled and normal child to parent propagation is in effect. The previous test showed that when enabled the query response was 122ms compared with 1.61s in this test. Please review the previous test results on page 54.

Test Function

```
Query: Find all the films Peter Sellers performed in along with the director's name and list all the actors and
the character they played in each film

func TestMoviePS3a(t *testing.T) {

        input := `{
  me(func: eq(name,"Peter Sellers") ) {
    name
    actor.performance {
         performance.film  {
                 title
                 film.director {
                         name
                 }
                 film.performance {
                         performance.actor {
                                 name
                         }
                         performance.character {
                                 name
                         }
                 }
         }
   }
 }
}`

        expectedTouchLvl = []int{1, 15, 15, 391, 744}
        expectedTouchNodes = 1166

        stmt := Execute("Movies", input)
        result := stmt.MarshalJSON()
        t.Log(stmt.String())

        validate(t, result)

}
```

$ go test -run=TestMoviePS3a -v

```
gql: 2021/04/05 22:31:47.368051 Duration: Parse  83.421952ms  Execute: 1.610042469s
```

See JSON output in previous test

Log Output

```
DB:2021/04/05 22:31:45 log.go:89: ================================== SetLogger
gqlES: 2021/04/05 22:31:45.673948 Client: 7.9.0
gqlES: 2021/04/05 22:31:45.673977 Server: 7.8.1
gql: 2021/04/05 22:31:45.673991 Startup...
monitor: 2021/04/05 22:31:45.674035 Powering on...
grmgr: 2021/04/05 22:31:45.674049 Powering on...
gql: 2021/04/05 22:31:45.674060 services started
TypesDB: 2021/04/05 22:31:45.674650 db.getGraphId
TypesDB: 2021/04/05 22:31:45.743634 getGraphId: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 1 Duration: 68.891098ms
TypesDB: 2021/04/05 22:31:45.743693 db.loadTypeShortNames
TypesDB: 2021/04/05 22:31:45.747590 loadTypeShortNames: consumed capacity for Query: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 5 Duration: 3.836628ms
TypesDB: 2021/04/05 22:31:45.752694 LoadDataDictionary: consumed capacity for Scan: {
  CapacityUnits: 0.5,
  TableName: "GoGraphSS"
},  Item Count: 15 Duration: 4.603517ms
gqlDB: 2021/04/05 22:31:45.766675 GSIS:consumed capacity for Query index P_S, {
  CapacityUnits: 0.5,
  TableName: "DyGraphOD"
}.  ItemCount 1  Duration: 8.619904ms
DB FetchNode: 2021/04/05 22:31:45.766771  node: eneJBCenT1qrdnv4X//RLw== subKey: A#
DB: 2021/04/05 22:31:45.772582 FetchNode:consumed capacity for Query  {  <= Node Query
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 4  Duration: 5.761901ms                                  <= 4 items fetched in 5.76 ms
DB FetchNode: 2021/04/05 22:31:45.772930  node: BnzFhcj1THeM1+qX3qUlrw== subKey: A#G# <= Propagated data only
DB: 2021/04/05 22:31:45.779757 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 6.777053ms                                  <= 7 propagated items fetched in 6.77 ms
DB FetchNode: 2021/04/05 22:31:45.779891  node: FZBl1d6RSuGcDFlf4khLZQ== subKey: A#G#
DB: 2021/04/05 22:31:45.785687 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 5  Duration: 5.749351ms
DB FetchNode: 2021/04/05 22:31:45.785795  node: imUjsX0TQza3kZAEnqn88g== subKey: A#G#
DB: 2021/04/05 22:31:45.792246 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 6.411359ms
DB FetchNode: 2021/04/05 22:31:45.792382  node: BhP8DIdtQROQp8euYg7eMw== subKey: A#G#
DB: 2021/04/05 22:31:45.796782 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.351894ms
DB FetchNode: 2021/04/05 22:31:45.796910  node: 76k8rK8WR2qiKxaKaXARrg== subKey: A#G#
DB: 2021/04/05 22:31:45.801587 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.60524ms
DB FetchNode: 2021/04/05 22:31:45.801772  node: Pdnun4yURwabuYON0emKCw== subKey: A#G#
DB: 2021/04/05 22:31:45.806297 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.45094ms
DB FetchNode: 2021/04/05 22:31:45.806520  node: 9VF0b6nIRsab6lRmgz5DEA== subKey: A#G#
DB: 2021/04/05 22:31:45.811010 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.438765ms
DB FetchNode: 2021/04/05 22:31:45.811269  node: fQ04rnM9TjmLVu+KUUrrow== subKey: A#G#
DB: 2021/04/05 22:31:45.815853 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.531328ms
DB FetchNode: 2021/04/05 22:31:45.815994  node: BnzFhcj1THeM1+qX3qUlrw== subKey: A#G#
DB: 2021/04/05 22:31:45.819420 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.380928ms
DB FetchNode: 2021/04/05 22:31:45.819565  node: dEXNfXIqQqCYjGAtbjYF/Q== subKey: A#G#
DB: 2021/04/05 22:31:45.824946 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
```

```
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 5.332675ms
DB FetchNode: 2021/04/05 22:31:45.825196  node: DPVhzClFReGftE5z/mY6Sw== subKey: A#G#
DB: 2021/04/05 22:31:45.831113 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 5.861799ms
DB FetchNode: 2021/04/05 22:31:45.831419  node: owlXqxNITf6PeDwNkmAi2A== subKey: A#G#
DB: 2021/04/05 22:31:45.837304 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 5.811945ms
DB FetchNode: 2021/04/05 22:31:45.837519  node: YWOEQhLoRlKWWTHARtqejQ== subKey: A#G#
DB: 2021/04/05 22:31:45.842090 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.49117ms
DB FetchNode: 2021/04/05 22:31:45.842227  node: k/4DFiGiQl21KIN++n5ApQ== subKey: A#G#
DB: 2021/04/05 22:31:45.847458 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 5  Duration: 5.175955ms
DB FetchNode: 2021/04/05 22:31:45.847655  node: q4ud/3POQBivuGKeVuvEXA== subKey: A#G#
DB: 2021/04/05 22:31:45.851123 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.420601ms
DB FetchNode: 2021/04/05 22:31:45.851248  node: LTGTzyBBSKin2KUPCGTRPQ== subKey: A#G#
DB: 2021/04/05 22:31:45.855850 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.558105ms
DB FetchNode: 2021/04/05 22:31:45.855984  node: e83avEkoQFiZo1fWVKUqTA== subKey: A#G#
DB: 2021/04/05 22:31:45.860046 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.011843ms
DB FetchNode: 2021/04/05 22:31:45.860489  node: AjOBJ9xlRRyDrBzRyMy+Xg== subKey: A#G#
DB: 2021/04/05 22:31:45.864930 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.394858ms
DB FetchNode: 2021/04/05 22:31:45.865036  node: 66sNzWLqQaCJkwO29mlqRQ== subKey: A#G#
DB: 2021/04/05 22:31:45.870679 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 5.600753ms
DB FetchNode: 2021/04/05 22:31:45.870784  node: BXFnHue0T/6UVY3DeBDMHA== subKey: A#G#
DB: 2021/04/05 22:31:45.874238 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.407276ms
DB FetchNode: 2021/04/05 22:31:45.874483  node: 0OiknU6VQfqAxqkUz+Y3vw== subKey: A#G#
DB: 2021/04/05 22:31:45.877697 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.162906ms
DB FetchNode: 2021/04/05 22:31:45.877824  node: Ul+IxXnHTsavwUKRH9PEvg== subKey: A#G#
DB: 2021/04/05 22:31:45.882406 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.536243ms
DB FetchNode: 2021/04/05 22:31:45.882539  node: nkDNMPYGSBSnRgsX0b2djA== subKey: A#G#
DB: 2021/04/05 22:31:45.886219 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.63708ms
DB FetchNode: 2021/04/05 22:31:45.886330  node: Bd9CVfTeR2+eSPTI/iaRmw== subKey: A#G#
DB: 2021/04/05 22:31:45.890296 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.919437ms
DB FetchNode: 2021/04/05 22:31:45.890421  node: APdVuKvIQAS/2KaXoMjhOA== subKey: A#G#
DB: 2021/04/05 22:31:45.893793 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.329915ms
DB FetchNode: 2021/04/05 22:31:45.893917  node: OB/DBDt3SlKhjlowjFxQQA== subKey: A#G#
DB: 2021/04/05 22:31:45.898262 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.303241ms

. . . .

DB FetchNode: 2021/04/05 22:31:46.187904  node: 02W0DG9TQ8iKKs4vV9m1bw== subKey: A#G#
DB: 2021/04/05 22:31:46.192385 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.431739ms
DB FetchNode: 2021/04/05 22:31:46.192623  node: 0l3jc2RfRUaIEHX85bmGpQ== subKey: A#G#
```

```
DB: 2021/04/05 22:31:46.196830 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.16039ms
DB FetchNode: 2021/04/05 22:31:46.197089  node: gG1g4BmVQA6dMmkQMhc5Ug== subKey: A#G#
DB: 2021/04/05 22:31:46.200507 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.358192ms
DB FetchNode: 2021/04/05 22:31:46.200670  node: tYQ19sZlTDW6lKYZCSQG7Q== subKey: A#G#
DB: 2021/04/05 22:31:46.204159 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.449152ms
DB FetchNode: 2021/04/05 22:31:46.204346  node: r78U68GPQLWpZF21IAqi5Q== subKey: A#G#
DB: 2021/04/05 22:31:46.208511 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.116352ms
DB FetchNode: 2021/04/05 22:31:46.208729  node: MWcKVKRYQ4WH7gP954orCw== subKey: A#G#
DB: 2021/04/05 22:31:46.212654 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.866351ms
DB FetchNode: 2021/04/05 22:31:46.213047  node: sOxHlO/ETj2JOVuGO7t+mg== subKey: A#G#
DB: 2021/04/05 22:31:46.216580 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.447009ms
DB FetchNode: 2021/04/05 22:31:46.216847  node: 4NlLo+tGT76nubRXnscZCw== subKey: A#G#
DB: 2021/04/05 22:31:46.220302 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.408951ms
DB FetchNode: 2021/04/05 22:31:47.326028  node: sii7dfM6TwGoFO3C8GxAvw== subKey: A#G#
DB: 2021/04/05 22:31:47.329682 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.58917ms
DB FetchNode: 2021/04/05 22:31:47.329846  node: VV2v0wrdQleKGcN1dlE8DQ== subKey: A#G#
DB: 2021/04/05 22:31:47.334625 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.735487ms
DB FetchNode: 2021/04/05 22:31:47.334768  node: fEhFAq5MTEmCGdvNA6h2dg== subKey: A#G#
DB: 2021/04/05 22:31:47.338794 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.976725ms
DB FetchNode: 2021/04/05 22:31:47.338959  node: yx7Ps+LcQ7WNKh6erqjn0Q== subKey: A#G#
DB: 2021/04/05 22:31:47.342810 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.766058ms
DB FetchNode: 2021/04/05 22:31:47.342952  node: SjqmklYPQseA64rf51EuNA== subKey: A#G#
DB: 2021/04/05 22:31:47.346280 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.286883ms
DB FetchNode: 2021/04/05 22:31:47.346397  node: kIS5FtuYTwKNPqInv/Fa5A== subKey: A#G#
DB: 2021/04/05 22:31:47.350928 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 4.491374ms
DB FetchNode: 2021/04/05 22:31:47.351115  node: FHQgpkSvTDCbEEPESKZbXg== subKey: A#G#
DB: 2021/04/05 22:31:47.354917 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.757904ms
DB FetchNode: 2021/04/05 22:31:47.355044  node: 3AkDImd+Qx+/B2mkdsL3ag== subKey: A#G#
DB: 2021/04/05 22:31:47.358707 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.55996ms
DB FetchNode: 2021/04/05 22:31:47.358866  node: dIPJwmoDRv6vTkYwsUQfCw== subKey: A#G#
DB: 2021/04/05 22:31:47.363957 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 5.041792ms
DB FetchNode: 2021/04/05 22:31:47.364091  node: JbRGSVEIR/u+ogbgSTx7Zg== subKey: A#G#
DB: 2021/04/05 22:31:47.367937 FetchNode:consumed capacity for Query  {
  CapacityUnits: 1,
  TableName: "DyGraphOD"
}. ItemCount 7  Duration: 3.7967ms
gql: 2021/04/05 22:31:47.368051 Duration: Parse  83.421952ms  Execute: 1.610042469s
monitor: 2021/04/05 22:31:49.373512 monitor: []interface {}{1, 1, 1166, []int{1, 15, 15, 391, 744}, 1166, []int{1,
15,  15,  391,  744},  359,  (*monitor.Fetch)(0xc0002faba0),  interface  {}(nil),  interface  {}(nil)}
monitor.Fetch{Fetches:359, CapacityUnits:359, Items:2484, Duration:1507493914}
DB: 2021/04/05 22:31:49.444505 TestLog: consumed capacity for PutItem  {
  CapacityUnits: 68,
  TableName: "TestLog"
```

}. Duration: 15.391477ms

Appendix A : CSP Programming in Go

To write a CSP (Communicating Sequential Process's) program in Go you need to be aware of these key concepts:

goroutines

Goroutines are normal functions that are made to run asynchronously to the calling routine be it the main program (aka the *main goroutine*) or another *goroutine,* by prefixing its instantiation with the “go” keyword. For example, to run function *foo(bar)* or a function literal*,* synchronously and asynchronously with the main routine:

```
func main() {
var x int
var y string
Var. c bool = true
…
// call foo synchronously

foo(bar)
…
// call foo asynchronously

go foo(bar)

// call a function literal asynchronously

go func() {….}()

// pass in arguments to a goroutine

go func(a int, b string) {….}( x, y )
```

The first instantiation of *foo* will run serially with the main program, whereas the second instantiation, prefixed with “go”, will run *foo* asynchronously with the main program. Both *foo* instantiations will have access to *boolean c* as they are closures (all Go functions are closures), however as c is a shared variable and if either *main* or *foo* modifies c, extra precautions will need to be made describe in the “shared variables” section below.

While *goroutines* provide a computing component they are useless unless they can communicate their output to other computing components. Channels are the mechanism *goroutines* use go pass data and synchronise events between other *goroutines*. Channels are the subject of the next section.

channels

While *goroutines* represent the SP (sequential processes) in CSP, channels represent the mechanism for the C (communicating). Channels are the way we link producers of data (SP) to consumers of the data. The producer writes some data onto the channel and the consumer reads that data from the channel using the syntax that is describe below. Channels are also used to synchronise events between produces and consumers. How do they do that? The write to a channel will always block if no consumer is listening on the channel. Similarly the consumer will always block when listening on the channel if no producer has written to the channel. Consequently when a producer writes to a channel without blocking we know that the produce and consumer are synchronised at that point in the code. Channels can also be defined with a buffer of any size, which will delay the synchronisation behaviour until the buffer has been filled.

To give you a feel for the elegant syntax that is Go channels, the following examples provide a basic overview, however there is more to Channels than presented here.

To define a channel called *myChan* that can accept integer data, declare the following:

```
var myChan int 5

// must make a channel before using it.
// second argument is the channel buffer value, yes a channel can b a queue
```

```
myChan = make(chan int, 0)
```

Channels can pass almost any data, *scalar* values, *slices*, *arrays*, *structs*, *pointers*, *channel* values, *function* values.

To write and read to a channel we use the symbol "<-" placed on the right or left of the channel variable respectively.

Lets write the value 5 to the channel:

```
a:=5

myChan <- a      // goroutine will wait here until the receiving goroutine reads from the channel
                 // alternatively, writing to the channel will unblock a waiting reading goroutine
```

Another *goroutine* can read from the channel using either of the following syntax:

```
for b := range myChan {…}     // often used in pipelined goroutines
```

or

```
b := <- myChan                // common way to synchronise and pass data
```

or

```
<- myChan                     // ignore data and synchronise goroutines instead
```

The myChan value, for the reading *goroutine*, might be passed in as an argument or passed via a channel.

Traditional system programming languages like *C* and *Java* use shared memory, provided by the OS, to facilitate data sharing and communication between concurrent programs. *Go* on the other hand, uses channels to pass data and synchronisation events between *goroutines*. This leads to one of Go's principal mantra's that I should mention at this point "*don't communicate by sharing memory, share memory by communicating*".

Don't be afraid to be generous with the use of channels. Channels are light weight components. Many applications require each instance of data in an array or slices to have its own channel for example.

shared variables

g*oroutines,* like all Go functions, are closures.

Variables created within a function are necessarily private to the function, variables declared outside the function represent *shared data* that can be accessed by other concurrent *goroutines* that have the same data scope. If the shared data is only ever read by the concurrent *goroutines* there will be no concurrency issues that can cause data corruption. However, if the shared data is modified by one or more concurrent *goroutine* special precautions need to be made, as concurrent read and update operations on the same data can lead to data corruption. If such precautions have been made the associated package that instantiates the concurrent *goroutines* is said to be *concurrency safe.*

Go provides two approaches to achieving *concurrency safe* programs*.*

One approach uses the *sync package API* (e.g. *Lock() & Unlock()*) to synchronise access to shared data provided all functions that access the data use the API.

Another approach encloses the shared data in its own *gofunction.* Any function requiring access to the data (both read or update) is forced to use a public accessibly channel to do so. In this way channels are used to serialise access.

To validate your *goroutines* are **concurrency safe** add the *-race* option when testing your code, e.g:

```
go test -run=parallelLoad -race -v
```

This will fail the test if an unsafe operation occurs on any shared data at runtime along with detailed diagnostic output. The *-race* option is not foolproof, however, as it will only fail a test if an actual race condition occurs, and by definition race conditions don't always occur. Consequently you need to run your tests multiple times if there is a chance it may experience a race condition, and even then unsafe code may not be caught during testing.

If you don't specify the *-race* option Go will PASS a test even if a race condition occurred. There's an overhead to using the option so only use it when your using concurrent routines that perform update and reads on the some shared data.s

Sync Package

I refer you to the WaitGroup type in the Go package documentation: https://golang.org/pkg/sync

Appendix B - How AWS Charges for Dynamodb?

For the best description of what factors affect the costs of using Dynamodb I cannot do better than the AWS documentation.

Read Consistency

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

Read Write Capacity Units

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html

Appendix C - Scalability and shared resources

Following on the from discussion on SQL vs NoSQL, ...

Any in-memory resource, be it a cache of any sort, a list structure of any sort, or just a single variable, that is accessible by two or more concurrent processors, must be protected against one process mutating the resource while another process reads it. Why does it need to be protected? Because a reading process is likely to get corrupted data while the resource is being mutated. To prevent the potential for corrupted reads all access to a shared resource must be serialised.

To serialise access to a shared resource there is an in-memory object called a **mutex** which acts as a kind of gatekeeper to the resource. Any process requesting to read or update a shared resource must first make a request to the mutex for access. The mutex will either grant access or block the requestor forcing it to wait (measured in microseconds) before it can try again, and therein lies a potential scalability issue. Increase the frequency of the requests and the waits on the requesting processes will increase.

So any **shared** resource that can be mutated has the potential to be a scalability issue. The factors that determine the potential are:

* how often is the resource accessed in the code
* is the access required in all, or often executed, code paths?
* the number of concurrent processors requesting access
* the frequency of that access from the database session

So to highlight this issue with a real world Oracle database scenario I will refer you to a Telco Billing product I was associated with many years ago. It was entirely written in Oracle's PL/SQL and at the time ran in a single Oracle instance on small to large SMP servers with 4 or more CPUs. The Rating Engines (RE) component would process hundreds of millions of Call Data Records (CDRs) a day and the server would usually be allocated a RE per CPU. However, as the number of REs increased beyond 4 there would be a measurable degradation in CDR throughput per RE. For the larger servers with up to 30 cpus, the degradation would continue to the point where adding an RE would make no difference to the overall CDR throughput.

Enter, Oracle's comprehensive and exquisitely detailed instrumentation of the database and the STATSPACK utility that aggregates this data by discrete time intervals (aka snapshots) and allows the user to generate a very detailed "stats report" between any two snapshots. One of the top level reports listed the top five "event waits" in database experienced for that interval. In a properly running database all the top five "events" would be related to some type of physical IO event, such as a "log sync", "sequential read", "scattered read" etc. However, when the REs were running one of the top five events was related to a mutex wait on the library cache. The "library cache" is a shared resource that maintains the parsed information of each SQL statement. So it was a surprise to see a mutex wait in the top five and even more of a surprise to find it consumed 24% of the overall event waits in the database. Why?

The problem was the REs. Each RE issued about 20 to 30 different SQL statements to the database on a continual basis. Each Oracle background process associated with an RE, would first generate a hash on the SQL statement and check if it existed in the library cache. So in answer to the four factors listed above to determine the potential for a scalability issue it it rates very high to extreme

* how often is the resource accessed in the code. **For every SQL statement issued to database.**
* is the access required in all, or often executed, code paths? **All**
* the number of concurrent processors requesting access. **Same as number of REs** (CPUs)
* the frequency of that access from the database session. **Extremely high - CPU speed**

The most significant of these is the last factor. REs generate SQL calls at CPU speed rather than the usual OLTP speeds of human operators. Consequently the pressure on the library cache mutex is extremely high to the point that each concurrent Oracle process is waiting hundreds of milliseconds to access the library cache rather than the usual tens to hundreds of microseconds of an OLTP application. As a result of these

mutex waits the REs where effectively throttled by 24% and the waits would only increase as more REs were added.

How did we resolve this disastrous scalability issue. There were two solutions. Either make the RE architecture distributed so there would be multiple Oracle instances with 2 to 4 REs per instance. This put less pressure on the mutex in each instance. The second solution relied upon the fact that the RE data was static when the REs was running. Updates to the data would only occur when the REs were shutdown. So instead of relying upon the Oracle data cache, each RE would load the static data (across perhaps 25 tables) into local memory (a PL/SQL table for each physical table) during startup. The downside of this solution was it took a long time to load some PL/SQL tables which seriously impacted startup times and PL/SQL tables are very very memory hungry. The final solutions was a merging of both. Make RE a distributed database application using commodity based servers and for heavily accessed tables cache them in application memory. Scalability issue resolved and performance was outstanding.