Poseidon: A OneGraph Engine

Brad Bebee
Amazon Web Services
Seattle, Washington, USA
beebs@amazon.com

Ümit V. Çatalyürek*
Amazon Web Services
Atlanta, Georgia, USA
uvc@amazon.com

Olaf Hartig†
Amazon Web Services
Linköping, Sweden
olaf.hartig@liu.se

Ankesh Khandelwal
Amazon Web Services
Seattle, Washington, USA
ankeshk@amazon.com

Simone Rondelli
Amazon Web Services
Miami, Florida, USA
rondelli@amazon.com

Michael Schmidt
Amazon Web Services
Seattle, Washington, USA
schmdtm@amazon.com

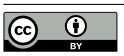
Lefteris Sidirourgos‡
Amazon Web Services
Berlin, Germany
sidirol@amazon.com

Bryan Thompson
Amazon Web Services
Seattle, Washington, USA
bryant@amazon.com

Abstract

We present the Poseidon engine behind the Neptune Analytics graph database service. Users interact with Poseidon using the declarative openCypher [11] query language. It enables requests that seamlessly combine traditional querying (such as graph pattern matching, variable length paths, aggregation) with graph algorithm invocations and has been syntactically extended to facilitate *OneGraph* interoperability, i.e., the disambiguation between globally unique IRIs (as exposed via RDF) vs. local identifiers (as encountered in LPG data). Poseidon supports a broad range of graph workloads, from simple transactions, to top-k beam search algorithms on dynamic graphs, to whole graph analytics requiring multiple full passes over the data. For example, real-time fraud detection, like many other use cases, needs to reflect the current committed state of the dynamic graph. If a user's cell phone is compromised, then all newer actions by that user become immediately suspect. To address such dynamic graph use cases, Poseidon combines state-of-the-art transaction processing with novel graph data indexing, including lock-free maintenance of adjacency lists, secondary succinct indices, partitioned heaps for data tuple storage with uniform placement, and innovative statistics for cost-based query optimization. The Poseidon engine uses a logical log for durability, enabling rapid evolution of in-memory data structures. Bulk data loads achieve more than 10 million property values per second on many data sets while simple transactions can execute in under $20\mu s$ against the storage engine.

*Ümit V. Çatalyürek is appointed both as an Amazon Scholar and as a Professor at Georgia Institute of Technology. This paper describes work performed at Amazon.
†Olaf Hartig is appointed both as an Amazon Scholar and as a Senior Associate Professor at Linköping University. This paper describes work performed at Amazon.
‡Corresponding author



SIGMOD Companion '26, Bengaluru, India

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803076

CCS Concepts

• **Information systems → Graph-based database models**.

Keywords

graph databases; data storage; graph algorithms;

ACM Reference Format:
Brad Bebee, Ümit V. Çatalyürek, Olaf Hartig, Ankesh Khandelwal, Simone Rondelli, Michael Schmidt, Lefteris Sidirourgos, and Bryan Thompson. 2026. Poseidon: A OneGraph Engine. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3788853.3803076

1 Introduction

Since its launch in 2017, thousands of users have created tens of thousands of Amazon Neptune graph data clusters because of the choice of data models and query languages (Labeled Property Graph (LPG) [23, 25] or Resource Description Framework (RDF) [8]), enterprise features, and the simplicity of a fully managed graph database experience. However, users have asked for LPG/RDF interoperability, higher load rates, better low latency query performance, and graph analytics support. To address these user needs we launched Neptune Analytics, a service providing graph query, graph algorithms, and graph analytics, based on a new engine, *Poseidon*. The Poseidon engine, designed for the *OneGraph* (1G) data model [15], directly addresses interoperability in the data between RDF and LPG data [28] and fully supports both graph representations, including LPG concepts such as multiple edges for the same source and target vertex, meta-properties, edge properties, etc.

With Neptune Analytics, users can get insights and find trends by processing large amounts of graph data in seconds. To analyze graph data efficiently, Neptune Analytics stores large graph datasets in memory. It supports a library of optimized graph analytic algorithms, low-latency graph queries, and vector search capabilities. They use Neptune Analytics for investigatory, exploratory, or data-science workloads that require fast iteration for data, analytical and algorithmic processing, or vector search on graph data. It complements Amazon Neptune Database, a popular managed graph

database. To perform intensive analysis, users can load the data from a Neptune Database graph with a snapshot into Neptune Analytics, or alternatively load graph data that's stored in Amazon S3.

Currently, users interact with Neptune Analytics using the declarative openCypher [11] query language, which enables requests that combine traditional querying paradigms (such as graph pattern matching, variable-length paths, aggregation) with algorithm invocations and has been syntactically extended to facilitate *OneGraph* interoperability, such as the disambiguation between globally unique IRIs (as exposed via RDF) vs. local identifiers (as encountered in LPG data). Poseidon supports a broad range of graph workloads, from simple transactions, to top-k beam search algorithms on dynamic graphs, to whole graph analytics requiring multiple full passes over the data. For example, real-time fraud detection, like many other use cases, needs to reflect the current committed state of the dynamic graph. If a user's cell phone is compromised, then all newer actions by that user become immediately suspect. To address such dynamic graph use cases, Poseidon combines state-of-the-art transaction processing with novel graph data indexing, including lock-free maintenance of adjacency lists, succinct secondary indices, partitioned heaps for data tuple storage with uniform placement, and innovative statistics for cost-based query optimization. The Poseidon engine uses a *logical log* for durability, enabling rapid evolution of in-memory data structures. Bulk data loads achieve more than 10 million property values per second on many data sets while simple transactions can execute in under $20\mu s$ against the storage engine.

Poseidon is a main memory, high performance, low latency, hybrid transactional and analytics (HTAP) graph storage engine, built from scratch to extend Neptune Database's disk-based architecture with in-memory analytics and HPC-style parallelism. Its hybrid columnar design is heavily influenced by advances in a) High Performance Computing (HPC) graph algorithms, b) Main-Memory Databases for efficient data retrieval, and c) Transactional Databases for supporting demanding online transactional workloads. The design balances performance requirements for *graph algorithms*, as well as for *OLAP* and *OLTP* queries. To achieve this goal, we designed and built a hybrid in-memory data storage engine with graph-specific partitioning and indices. Poseidon maintains the locality of specific graph access paths: the property set of a vertex is always within a single partition, and the edge properties are *co-located* in the same *container* as the edges that they describe. In addition, data representing the *topology* of the graph (i.e., edges connecting resources) is distributed across partitions, which ensures that we can scale graph algorithms when the input frontier becomes large. While the Poseidon engine does well on both OLTP and OLAP workloads, OLAP workloads can also be run on separate compute to minimize the impact on transactional workloads.

The graph is organized as a system of partitioned relations for string data, vertex properties, edges, edge properties, etc. Internally, each partition maintains a heap of narrow tuples on PAX pages [1]. Secondary indices are maintained over that heap. Graph-specific indices and lock-free data structures are used throughout to achieve peak performance against a highly dynamic transactional graph. The query engine and algorithm kernels interact with an *Access Pattern Language* (APL) which decouples the internal organization of memory and provides low-level data access optimizations. The engine is also decoupled from the durable representation using a logical log, thus enabling very rapid evolution of the different components.

The contributions of this publication are:

(1) A system-level realization of the 1G abstract model, including the design of partitioned relations for modeling a 1G graph in Poseidon, the physical organization of tuples on PAX pages in memory, the use of secondary indices, etc.;
(2) A discussion of transactions and durability, including the use of fast *mvcc* and how using a logical log accelerates innovation;
(3) The introduction of the Access Pattern Language (APL) which provides a separation of concerns encapsulating access to the physical organization of the graph in memory, and how the APL selects the most appropriate low-level scan kernels for each access pattern expression;
(4) A discussion of graph algorithms in Poseidon, including how algorithms can run against live data or against static views from a point in time, how algorithm kernels are written and reused internally by the engine, dynamic switching between index-driven and scan-driven, and how algorithms are parallelized; and
(5) A sketch of key aspects of the query language and runtime query processing and algorithm integration, and description of the design principles behind the interaction of the cost-based query optimizer with storage-level APIs for cardinality estimation and higher-level statistics.

2 1G Abstract Model

The Neptune graph database service supports both the graph data model of RDF [8] and LPGs [23, 25], with their query languages SPARQL [13], and Gremlin [24] and openCypher [12], respectively. However, users still had to choose one of these models when creating a new database instance and could not "cross-use" these technologies (e.g., querying LPGs using SPARQL, or RDF using openCypher). We have observed that this limitation can cause confusion (especially, for users who are new to graphs) since making the choice is not trivial as it requires considering data modeling aspects, query language features, adequacy for current and futur e use cases, and developer knowledge. To remove these obstacles and achieve interoperability between these technologies we are pursuing the OneGraph (1G) project [15].

An initial result of this effort — available to users of Neptune Analytics, on top of Poseidon— is the option to load LPG and RDF data into a single, connected graph and to query this graph using openCypher [28]. The conceptual basis of this functionality is an abstract data model, called the *1G model*, which combines features from both LPGs and RDF and can be used as a foundation for switching seamlessly between an RDF and an LPG view of the data.

While presenting the formal definition of this model and relevant data and query language mappings is beyond the scope of this paper, we briefly summarize the main concepts of the model: A graph dataset in the 1G model consists of a set of *1G elements* that have an identity and can assume different roles within the dataset. In particular, every 1G element may represent a node in a

graph. Moreover, there can be 1G elements that represent *property statements* which resemble the notion of properties in LPGs or of RDF triples with literal objects. Other 1G elements of a 1G dataset may represent *relationship statements* which resemble edges in LPGs or RDF triples that have a resource as object. Every statement of either of these two kinds is associated with a 3-tuple consisting of a 1G element that is the *subject* of the statement, a 1G element that is the *predicate*, and a 1G element or value as *object*. Finally, every 1G dataset consists of graphs, which are containers of statements. An important feature of the model is that statements may consist of 1G elements representing other statements, which may be used to capture statement-level metadata and even meta-edges (e.g., edges between edges), similar to the RDF-star extension of RDF [14].

The abstract 1G model maps naturally to a 4-column relational representation in which the fourth column contains *statement identifiers* (*SIDs*) for the statements captured via the first three columns. Hereafter, these first three columns are named S, P, and O, respectively, as they contain the subject, the predicate, and the object of each statement, and the fourth column (with the SIDs) is named I. On the logical level, the domains of these columns can be RDF terms (IRIs, blank nodes, literals) [8], for which the graph-scoped notion of identity of elements in LPGs needs to be harmonized with the globally-scoped notion of Internationalized Resource Identifiers (IRIs) [9] in RDF. We use the notion of a *baseIRI* associated with each graph dataset, as discussed below. Note that multiple SPOI tuples with the same S, P, and O values are permitted by the 1G model; this supports the LPG concept of multiple edges having the same source, link type, and target. If a query engine that runs in SPARQL mode encounters such data, it must project out the distinct SPO values to reduce the data model to RDF triples.

3 1G Physical Data Model

The naive way to define a physical data model for 1G would be a simple 4-column relation, where each of the SPOI columns is represented in lexical form. Such a representation will not be memory efficient as it would necessitate replicating data in every column. Poseidon is designed to be an in-memory storage engine for both OLTP and OLAP workloads. Thus, data must be stored in a compact way, data locality must be maximized, and parallelism must be enabled for higher throughput and scale-out computation. To meet these requirements, we use dictionaries to encode literals (lexical or binary data) and IRIs, mapping them to 64-bit global unique identifiers (gui). We separate literals and IRIs into two different dictionaries, which allows for faster localized access of IRIs, since they are separate from literals that can be very large strings or blobs. Similarly, we separate 1G elements, such as property statements and topology statements, in different relations to further improve low-latency access via more compact and local relations. We also partition the relations to enable parallel access, while making sure that we co-locate topology and property relations of the same set of vertices to facilitate joins between them.

In total, Poseidon's storage model is based on 13 relations. Two *dictionary relations* which provide a mapping between lexical forms and internal 64-bit Global Unique Identifiers (*GUI*s). The two dictionary relations are Rel_Vert and Rel_Lit, for vertex identifiers and string literals, respectively. We have four *topology relations*: simple edges are modeled in Rel_E, and edges which connect meta-information in another family of relations are modeled separately within the relations Rel_Me, Rel_Lme, and Rel_Rme; six *property value relations*: one to model vertex property sets (Rel_Vp) and one for vertex meta-properties (Rel_Mvp), then one more for edge property sets (Rel_Ep), and finally three for edge meta-properties (Rel_Lmep, Rel_Mep, Rel_Rmep); a *vector relation*, Rel_Vss, which maps vertex identifiers to vectors and maintains a Hierarchical Navigable Small World (HNSW) [18] secondary index over those vectors for efficient approximate nearest neighbor search.

Table 1 presents the different types of relations (topology, values, meta properties). The resulting system of 13 distinct relations is grouped into logical containers. These containers provide the ability to *co-locate* certain data for efficient access. For example, edges and edge properties are located in conformal containers to provide efficient access as are vertex properties and meta-properties. Table 2 shows how relations are grouped in the same containers, and how they share *join indexes* between common SIDs.

For capturing a basic LPG, the following five relations are sufficient: Rel_Vert, Rel_Lit, Rel_Vp, Rel_E, Rel_Ep. The 1G model extends that by allowing *meta-edges* between edges, *meta-properties* both for vertex properties and for edge properties, etc. SPOI tuples for such meta-edges and meta-properties use the I value (i.e., the SID) of some other SPOI tuple in either the S or the O position. For efficient storage and access, we divide the edge-related meta information into three subgroups depending on whether the S and/or O are SIDs. Hence, 1G adds seven more relations (Table 1), three topology meta relations: Rel_Rme (S is a simple vertex, O is a SID), Rel_Lme (S is a SID, O is a simple vertex), and Rel_Me (both S and O are SIDs), three corresponding property value relations (Rel_Rmep, Rel_Lmep, Rel_Mep), and one more property value relation for the *meta-vertex properties* (Rel_Mvp). SPARQL named graphs (a subgraph container concept) are handled as meta-properties annotating edges and property values, as are user-defined edge identifiers.

In order to share the same system of vertex identifiers across LPG and RDF data within the (relational) 1G representation, we *logically* prefix each LPG vertex identifier with the *baseIRI* of each 1G graph. For example, if a customer has a graph `arn://xyz`, then the vertex identifier `"1"` is logically interpreted as `"arn://xyz/1"`. For the common case where identifiers are local IDs – as it always is the case for LPG data, which does not exhibit a built-in notion of global identifiers (as opposed to using the RDF aspects of 1G to refer to an IRI) – that prefix is compressed into a single bit in the gui created for the vertex identifier. All storage operations are performed on gui and the distinction between LPG and RDF identifiers is invisible to those operations. This approach not only aligns identifiers from these two models, but it also makes it possible to merge or federate LPG data by relying on the semantics of the RDF data model.

Having separate relations enables different partitioning schemes for them, as illustrated in Figure 1. For example, Rel_E is partitioned in 2D to optimize multi-level traversals, whereas Rel_Vp is partitioned in 1D to optimize locality for fetching vertex properties.

One of the difficult design decisions was how to handle edge weights. Efficient access to edge weights is critical for the performance of many graph algorithms, such as SSSP. However, both the LPG and RDF data models are schema flexible or schema never. In

Table 1: Relation descriptions, S and O column types, and statement identifier classification (SIDs)

Relations			S → O edge type	I (SID)
Topology:	*edges*	Rel_E	IRI/BNode → IRI/BNode	Sid_E
	left meta edges	Rel_Lme	SID → IRI/BNode	Sid_Lme
	meta edges	Rel_Me	SID → SID	Sid_Me
	right meta edges	Rel_Rme	IRI/BNode → SID	Sid_Rme
Edge Values:	*edge properties*	Rel_Ep	Sid_E, Sid_Ep → IRI/BNode, Literal	Sid_Ep
	left meta edge properties	Rel_Lmep	Sid_Lme, Sid_Lmep → IRI/BNode, Literal	Sid_Lmep
	meta edge properties	Rel_Mep	Sid_Me, Sid_Mep → IRI/BNode, Literal	Sid_Mep
	right meta edge properties	Rel_Rmep	Sid_Rme, Sid_Rmep → IRI/BNode, Literal	Sid_Rmep
Vertex Values:	*vertex properties*	Rel_Vp	IRI/BNode → IRI/BNode,Literal	Sid_Vp
	meta vertex properties	Rel_Mvp	Sid_Vp, Sid_Mvp → IRI/BNode, Literal	Sid_Mvp
	vertex vector search	Rel_Vss	IRI/BNode → vector	Sid_Vss
Dictionaries:	*vertices*	Rel_Vert	IRI/BNode mapped to gui	
	literals	Rel_Lit	String mapped to gui	

Table 2: Containers of co-located relations and join indexes

Container	Join Index
C_E=(Rel_E, Rel_Ep)	Rel_E.I ⋈ Rel_Ep.S
C_Me=(Rel_Me, Rel_Mep)	Rel_Me.I ⋈ Rel_Mep.S
C_Lme=(Rel_Lme, Rel_Lmep)	Rel_Lme.I ⋈ Rel_Lmep.S
C_Rme=(Rel_Rme, Rel_Rmep)	Rel_Rme.I ⋈ Rel_Rmep.S
C_V=(Rel_Vp, Rel_Mvp)	Rel_Vp.I ⋈ Rel_Mvp.S

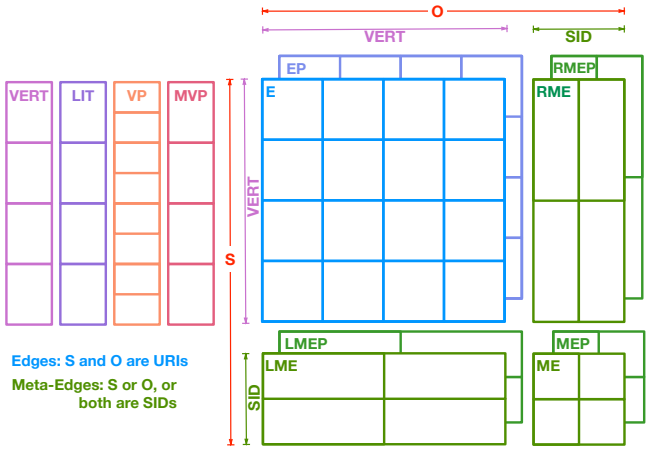


Figure 1: 1D and 2D partitioning of Poseidon relations.

our design, we decompose edges and edge weights into two different relations (Rel_E and Rel_Ep) which are conformally partitioned to ensure locality in scale-out architectures. We also define a join index to make it efficient to lookup the edge weights for an edge. Further, we surface the concept of Rel_E ⋈ Rel_Ep directly into the data access patterns, making it possible to push down optimizations into low-level data scan operations. This design allows us to properly model any customer data, including odd cases where an edge property is not always present, is present with the wrong data type, is present multiple times, is associated with meta-properties about that edge property, etc. To optimize performance for whole graph analytics, when it will be amortized, we build optimized data structures, such as CSR, which precomputes the edge and the edge weight for maximum performance.

In Poseidon, each partitioned relation consists of SPOI tuples (i.e., property statements and edge statements) that are stored in allocated memory heaps. The partitioned relations are organized as a set of 2MB huge pages using a PAX [1] format to support efficient scans. The tuples on each such page are managed by a lock-free free-list for that page. An Undo list is maintained per *zone* (a consecutive group of 1024 tuples), extending the concepts presented by Neumann et al. [21]. Secondary indices are also maintained over the partitioned relations using virtual arrays and lock-free adjacency lists. In addition, an *I-index* is defined that provides a lock-free factory for unique SIDs and a lookup of SPOI tuples by I.

To improve the in-memory data density, all data is stored compressed using 64-bit *Global Unique Identifiers* (gui) and, whenever possible, using only parts of the gui. We distinguish two kinds of gui identifiers: *dictionary gui*, as are assigned in the two dictionary relations (Rel_Vert, Rel_Lit), and statement gui that represent the tuple identifiers *SIDs*. Within each gui, we use four bits to encode which of the 13 relations the gui belongs to; for a dictionary gui, that is the dictionary relation in which the gui is assigned, and for a gui representing a SID, it is the relation containing the SPOI tuple in which that SID is the I value. The rest of the encoding depends on the kind of gui: A dictionary gui consists of a 44-bit identifier that is unique within the corresponding dictionary and is generated by Murmur hashing (16 bits in LSB positions) combined with a 28-bit collision counter, as illustrated in Figure 2. In contrast, a gui for SIDs consists of a 28-bit partition-local identifier (obtained from the partition-local I-index), plus 32 bits to identify the partition.[1]

[1]This means there can be a maximum of 2^{32} = 4B partitions and each partition can contain at most 2^{28} = 256M 1G elements. These limits come from the design choices based on our desire to limit the system recovery time, as well as to improve parallelism.

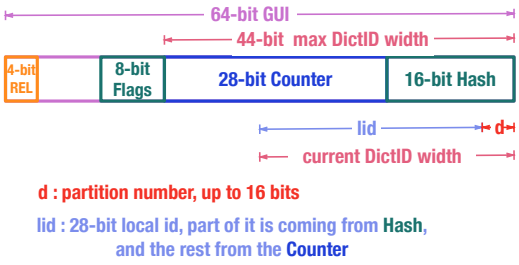


Figure 2: Dictionary encoding to gui.

Most relations use a 1D partitioning system where the low bits of the S determine the partition in which SPOI tuples for that S will be located (see Figure 2, where *d* bits are used for partition number) and all SPOI tuples for the same S are in the same partition. Furthermore, 2D partitioning is applied to the topology. This has several advantages (facilitating parallelism, load balancing, increasing the maximum degree of a vertex, minimizing contention in algorithm kernels), but also some trade-offs (adjacency lists of vertices are partitioned). The 2D partitioning system creates a grid of partitions where the rows of the grid are indexed by S and the columns of the grid are indexed by O. Thus, if only S is bound, then the adjacency list for that S is distributed across a row of the 2D grid. Likewise, if only O is bound, then the adjacency list for the O is distributed across a column of the grid. If both S and O are known, then the edges for that (S,O) are guaranteed to appear in a single partition. The 2D partitioning makes it possible to elide the low bits of each S and O value in the topology since those bits correspond to the row or column of the grid, thus we store S and O in 28 bits within Rel_E. Using similar logic, we store S in 28 bits within the partitions of Rel_Vp and the other non-meta relations (for meta relations we store the 64-bit S and O values). Also, all 1G relations have a partition local encoding of predicates, and hence the predicate is also stored in 4-bytes. The property value partitions use 9 bytes to store the O, including an 8-bit flags field, indicating type, and either inline primitive types up to 64-bits or the GUI of larger values. In total, topology (Rel_E) tuples are only 16-byte wide and property value tuples (Rel_Ep, Rel_Vp, etc.) are 21-byte wide.

4 Transactions and Durability

Poseidon supports a variant of "fast MVCC" [21], but with modifications to support a graph data model. Each zone (of 1024 rows) is associated with a version vector that marks tuples within that zone having UNDO information. The UNDO chain contains the pre-image version of the tuple. Due to their narrow width and the nature of the 1G data model, tuples are only inserted or deleted, not modified in place. Therefore, UNDO chains are at most 1- or 2-deep (insert only, delete only, or insert + delete). SPOI tuples are bit-marked if they are on the free list (a lock-free data structure which is intrusively linked for tuples on the same page) and if they are associated with multi-version information. Because the S and O values are based on a murmur hash and collision counter, neighboring values are not "close" in the application semantics. Therefore, rather than track *VersionedPositions*, we steer data access (scans) around versioned tuples using other mechanisms. Older versions are expunged once the Oldest Active Transaction (OAT) can no longer read a tuple.

All transactions are tracked in a transaction table. They are assigned a handle based on their position in the transaction table. A transaction is assigned a read timestamp and each successful read-write transaction also gets a commit timestamp. The timestamps are linearly ordered values stored in the transaction table, and the commit timestamp is incremental and unique. The durable logical log records are written incrementally after in-memory writes complete, and reference the transaction handle, which can be safely recycled after server restarts. A transaction abort or commit is recorded with the transaction handle, and the commit record also carries the commit timestamp. Incomplete transactions at the tail of the log are discovered on restart and explicitly aborted.

Undo logs are assigned to a chain of memory-aligned blocks, and each block is used exclusively for a single transaction. The transaction handle is stored once in the header of the blocks. When a SPOI tuple is read with multi-version bit set, we get the transaction handle in the header of the block and from there go to the transaction table to get the state of the transaction (in progress, committed, aborting, terminating, etc.) and commit timestamp for a committed transaction. By checking the state of the transaction and by comparing commit and read timestamps, we decide if a tuple is visible to a transaction.

We use another index to track read timestamps of actively running transactions and commit timestamps of committed transactions that need to be purged. The read timestamps of active transactions are reference counted. Whenever the reference count of a read timestamp becomes zero, we check whether we can advance the OAT. If the OAT is advanced, we purge any committed transactions with commit timestamp lower than the new OAT, including expunging any tuple versions that are no longer visible to any active transaction.

One of the early design decisions was to use a durable *logical* log. This has proven to be a tremendous benefit in developing a new storage engine because it completely separates the manner in which the data are organized for durability from the manner in which the data are organized for compute. This separation of concerns has made it possible for us to re-invent the in-memory layout and indexing methods on an ongoing basis, improving load rates, reducing memory stalls, reducing memory consumption, improving handling of hypersparse graphs, etc. without any concerns about breaking durable storage.

The durable log is fronted by a chain of lock-free box cars which write onto a partitioned log. Each partition of the log can accept up to a fixed write rate. By partitioning the log, we are able to scale the write throughput to the network bandwidth. We accept higher tail latency associated with a partitioned log because there is little locality in graph updates and, unlike relational workloads, nearly every query or update will touch multiple partitions.

Restore is from the most recent set of S3 checkpoints, followed by log replay. Like writes, restore is able to saturate the network bandwidth. Recovery and log apply are parallel and recovery time is bounded by the work each thread needs to do to restore its partitions.

5 Data Scan Operations

We support two methods of scan operations to retrieve data from storage. One is based on performing a complete scan over the relations heap, and the other through the use of indexes that map vertices to their adjacency list of edges. The choice between these two access methods – relation scan vs index scan – is based on a combination of factors: the type of relation (topology or values), the type of data (resources, strings, or inlined numericals), and the selectivity of the input filter. For example, if the vertices that are requested to be scanned have a high degree or if the query has a very low selectivity rate then we will opt for scanning the relation from the heap instead of going through an index scan. This is because we expect to read most of the data from the heap and produce many results, thus sequential scans will outperform index probes. On the other hand, if we are given a point query, or a small frontier of a set of vertices, then an index scan over the adjacency list index will be preferred. The input interface of all scan operations, including the filters, frontiers for joins, and projections, is based on the specialized language APL which we will detail in the next section. The APL implementation also contains the mechanism to decide whether to use a relation scan versus index scan, or even a combination of these two.

Relation Scans. As described previously, each relation is divided into partitions and each partition is stored in the memory heap. Each partitioned relation is divided into pages, where each page is an allocated 2 Megabyte huge page in memory. Consequently, each partitioned relation can have up to 2048 pages, and each page can have up to 131K SPOI tuples, totaling $2^{28} = 268$ million tuples per partition (for Rel_E with 16-byte wide tuples). Each page is then organized into a PAX [1] layout. The PAX blocking of a page, i.e. the way that columns are grouped together, depends on the type of relation, and also on the priority of the application (OLTP vs OLAP). We can block together all the columns for faster point queries, or break up columns into groups for faster sequential scans on a single column. For example, in the case of topology, all SPOI columns are grouped together in a 16-byte value to create an OLTP bias, while for value relations, we group together SPO and store the I column and the F column separately (the F column is used to mark the type of inlined values in O and is explained below).

Relation scans are sequential scans that follow the memory data layout (blocking) of the columns in order to achieve optimal memory access. Such scans maximize memory bandwidth reads because they facilitate pre-fetching and thus reduce cache and TLB misses. The relation column scan operation works by scanning each page of the partitioned relation following the blocking structure of the columns (i.e., the PAX layout) and by materializing the results for the current segment before proceeding to the next segment. Therefore, the relation scan algorithm can be viewed as a vectorized execution, where each vector is of the size of the PAX page.

We outline the relation scan algorithm for the general case, albeit small changes in the order of the columns scanned inside each page, subject to the different blocking layouts. The relation scan algorithm starts by looping over the pages in the partitioned relation. For each page, first the most selective column is scanned and a bit-vector is produced, such that the corresponding bits are set if the value in the scanned column matches the input filter or frontier. For the next column in the same page, we iterate over the bit-vector instead of the base data, and if the bit in position i is set, then we check the value in `col`$[i]$ if it matches the input filter and if so we continue, otherwise we set the bit i of the bit-vector to 0 and ignore that tuple. At the end, the algorithm has produced a bit-vector with bits set only for the SPOI tuples that satisfy the input filters. We use that bit-vector to materialize the results and continue to the next segment. In order for the relation scans to perform optimally, we avoid function calls and if-statement branching. For these reasons, our code relies heavily on C++ templates. The templates implement the access patterns as defined and enumerated by the APL interface. We also plan to extend our scan code with JIT compilation techniques to incorporate complex math computations, user defined functions, and aggregations with group by operations.

For the value relations where we are interested in the type of the literal values, which can be either strings or numerical, we introduce one more column, the Flag array that serves to identify the type of the O column. The Flag array has one 8-bit value per SPOI, effectively transforming SPOI to SPFOI. The purpose is to quickly identify the type of O during scan operations and thus skip over data that do not match the query filters or to quickly decide if a type can be promoted to another type, for example a short to integer. The Flag array divides literals in the O column not only according to their base type (e.g., integer, double, IRI, etc.) but also into larger groups of types, such as numerical vs non-numerical, composite vs non-composite, and also subgroups such as signed and unsigned, etc.

Index Scans. Besides relation scans, which are mostly useful for read-optimized queries, we also support point queries useful for transactional workloads. For this reason, we have defined three indexes over columns S, O, and I. For the S and O columns, the indexes are high-concurrency hashmaps of vertices mapped to adjacency lists, where lock-free updates are possible. For column S, the hash index maps a vertex identifier in all partitions, while for column O, the hash index is a vertex identifier for both the topology and wide topology partitions. For value relations, O is either a dictionary identifier for the case of string literals, or a resource (IRI). Notice that we do not include numerical values in the O index although in future work we plan to implement a specialized order-preserving index. The index on the I column is a join index that joins the co-located relations as described in Section 3.

The adjacency lists are lock-free data structures that maintain a direct list of all tuples in the partition having that dictionary identifier (gui) in the corresponding position. The adjacency list is a contiguous allocation with an exponential growth policy. When an adjacency list becomes full, it is compacted, deleted entries are purged (a delete marker is tracked), the size of the new adjacency list is determined, and the entries are copied over to the new adjacency list and then sorted again for physical locality. The sort order minimizes the likelihood that scanning an adjacency list induces TLB misses. The index of the last sorted position in the adjacency list is tracked and scan primitives can use it to optimize their behavior. The new adjacency list is installed into the hash index using an update/insert operation, which is based on a compare-and-swap instruction. In a data race, one thread wins and the other thread(s) discard their uninstalled version of the new adjacency list.

6 Access Patterns for 1G Data

Amazon Neptune is built with a collection of different components that can be stacked to build user applications. In order for Poseidon to be part of this software stack, we developed a data retrieval interface that is based on a declarative but frugal language. This language—called the *Poseidon access pattern language* (APL)—can be used to describe data retrieval requests over the logical SPOI relation used as the basis for representing 1G data in Poseidon (i.e., the language is agnostic to the physical data model). APL hides the internal data structures, index availability, and execution plans from the caller component, thus making Poseidon easy to integrate and use within Neptune. Given a data retrieval request, an APL processor creates a concrete data access plan for the request, runs this plan to fetch the requested data, and writes the results to a memory location specified as part of the request. To determine such a plan the APL processor employs a (rule- and cost-based) optimizer for making decisions about suitable access paths, and about usage of secondary indices such as bloom filters, imprints [32], etc. The remainder of this section provides an informal overview of APL and, thereafter, briefly describes the APL processor.

6.1 Poseidon Access Pattern Language (APL)

APL expressions are meant to be used as arguments for data retrieval operations—hereafter, called *APL requests*—that fetch data from the Poseidon relations. Additional arguments for such a request are pointers to data structures that either contain constants that are meant to be bound in a particular position of the specified access pattern (effectively supporting filters and frontiers) or represent the data structures into which the fetched data has to be written. Our design goals for APL are:

- APL should be able to *capture the range of access patterns* found across different query languages, while offering an extensible, query-language-agnostic abstraction layer.
- APL should be declarative, abstracting from both the separation of SPOI tuples into multiple relations on the physical level and the corresponding partitioning scheme.
- APL expressions should be *parsed efficiently* because parsing them is part of the critical path during query execution.
- APL expressions should be *very concise* and do not need to be easily readable for humans, because the language is designed to be generated by the query engine and interpreted by the storage layer.

Overall Structure of a APL Expression. Table 3 lists a few examples of APL expressions to provide an overview of the main features and the syntax of the language. Related to the four main attributes of the SPOI relations, APL expressions consist of four main components: the subject component (e.g., "S" in example expression #1 in Table 3), the predicate component ("P" in expression #1), the object component ("_^" in #1), and the SID component ("_" in #1). Within these components we use different annotation characters to define, e.g., if an attribute is to be projected or not in the retrieved data (the character to request projection is: ^), if it is unbound (character: _) or bound to a given set of constants (for filtering) or to a frontier (for joins against a known set of values), if a type flag is available (for exact type matching), etc. For APL expressions that define access patterns to retrieve data from multiple columns of the SPOI relations (e.g., example expression #2), the lists of values in the corresponding output data structures will be aligned with one another. That is, the i-th value in each of these lists will be obtained from the same SPOI tuple.

Filtering based on Values. An access pattern with a set of fixed values for a particular column can be specified by using the corresponding column name (e.g., "S" and "P" in example expression #1). Another option is to use the sub-expression "[]" (as in expression #2), which indicates that a list of possible values for the corresponding SPOI column is passed within the APL request and, then, considered in the access path as a frontier. The difference between these two options (e.g., "S" versus "_[]" in examples #1 and #2) is subtle: the first one refers to constant value(s) while the second to a frontier. Almost always, a frontier has to be evaluated with a (hash-)join, whereas constants can be handled either with joining or with sorting and probing, depending on the cardinality, statistics, and placement. Exactly these choices are meant to be hidden from the caller and left to the APL processor to decide. Notice that it is also possible to write "S[]" in order to indicate that both a set of constant values for filtering and a frontier are given.

Type Restrictions. For some cases, the query engine built on top of Poseidon may infer that the S or the O value of matching SPOI tuples can be restricted to a particular kind of values. For instance, if the query to be executed by the query engine retrieves edges that have edge properties, then the access pattern for this part of the query can be restricted to SPOI tuples that have a SID as their S value and a literal as their O value. Example #3 illustrates this case. The APL planner in Poseidon uses such type restrictions to prune irrelevant access paths (e.g., for example #3, only the relations Rel_Ep, Rel_Mep, Rel_Lmep, and Rel_Rmep are relevant).

Additional Sub-Pattern for the SID Component. In place of the SID component, an APL expression may contain another APL expression, as illustrated by example #4. Such types of combined APL expressions describe access patterns that resemble an equi-join with a predefined join condition (namely, the I value of the left-input tuples must be equivalent to the S value of the right-input tuples). The letter "x" between the two sub-expressions indicates the use of an inner join, whereas the letter "k" indicates a left outer join. The use case of such a combination of APL expressions is to capture typical patterns of joint data retrieval in a single APL request; e.g., retrieving edges together with their properties, or retrieving properties together with meta-properties about them. While a query engine on top of Poseidon may capture such joins also in its query plans, supporting them within a single APL request can result in more efficient query plans because Poseidon may, e.g., exploit partition co-location for such joins.

6.2 APL Planner

APL defines a finite number of different access patterns for retrieving data. However, these access patterns can entail different execution plans. The most common approach is to use statistics to predict the selectivity of an access pattern and thus choose index or relation scans. However, we can also employ heuristics in order to

Table 3: Examples of APL expressions, including the additional arguments expected when using them, and their meaning.

#	Expression	Meaning	Additional Arguments
1	`SP_^_`	retrieve the O value of every SPOI tuple that has a given S value and a given P value	i) value for S, ii) value for P, iii) pointer to data structure to which the O values of the selected tuples are written
2	`_[]P_^_^`	retrieve the O and the I value of all SPOI tuples that have both a specific P value and an S value that is one of the elements in a list of such values	i) value for P, ii) pointer to data structure with frontier for S; iii–iv) pointers to two data structures to which the O and the I values of the selected tuples, respectively are written
3	`s_^_l__`	retrieve all SIDs that are S values of SPOI tuples with a literal in the O position	i) pointer to data structure to which the S values of the selected tuples are written
4	`SP_^x__^_^_`	join every SPOI tuple (s, p, o, i) with every SPOI tuple (s', p', o', i') such that $i = s'$, and retrieve o, p', and o' of every result tuple	i–iii) pointers to three data structures to which the three values of every result tuple are written, respectively

decide the order in which we scan a column. Such heuristics have been studied in [34] and are also applicable in our design.

The APL planner first determines which partitions need to be scanned, subject to the existence of bounds on S and on O. The planner then determines if a relation or index scan will be used per partition, considering the cardinality of the set of vertices that can be found in each partition. Finally, the order of column access in the case of relation scans, or the usage of S or O index in the case of index scan is decided. In some cases, the planner may also choose to use both scan options and combine the intermediate results to reduce the data accessed or to first evaluate property predicates or inline values and then the adjacency lists of a set of vertices.

In addition, the APL planner will inject a *gather* operation if needed, in order to collect intermediate results from different partitions and compute a sum, or group together edges with a group-by parameter different from the indexed vertex. Finally, the planner will stop the execution of a scan if a Limit or Sample/Count operation is present in the query.

7 Graph Algorithms

Neptune Analytics provides a managed service for customers to run top-k (BFS, EgoNet, etc.), whole graph single pass algorithms (e.g., BFS, SSSP, WCC, etc.), and whole graph analytics (PageRank, Louvain, etc.) over live and dynamic graph data and achieve performance that is comparable with the state-of-the-art HPC graph analytics implementations.

Developing high-performance parallel graph algorithms is a challenging task that requires addressing several well-known hardware and software-related challenges [16]. Graph algorithms have random data access patterns, and since the ratio of computation to data access is very low, they are memory-bound. Furthermore, graph algorithms are data-driven and most of the real-life graph datasets have very skewed distributions. This causes a significant load imbalance challenge during algorithm execution. Literature shows that use of 2D partitioning can provide a sweet-spot to address some of these challenges [35], if not all. Hence, as described in Section 3, in Poseidon we adopted 2D partitioning of the topology relations. In each partition, SPOIs stored in our heaps can be viewed as Coordinate List (COO) format, which is one of the common formats in graph analytics, since each edge's two "coordinates", that is, S(ubject) and O(bject) are stored explicitly. Even though this could be an effective storage for some of the graph algorithms, especially on GPUs for edge-centric execution, most of the CPU-based HPC graph analytics systems (e.g., [31]) and individual algorithm implementations (e.g., [6, 36]), including recent baseline benchmark implementations (such as GAPBS [3]), work on static Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) representations of the graph, which are among the most compact and most efficient structures to access the adjacency lists of the vertices.

The interplay of data layout, dynamic data, and algorithm characteristics and their effect on performance is important to consider. For example, top-k analytics and algorithms that will do only a single pass over the graph run against the most recently committed state of the graph. This enables a wide range of applications where the recent information can be critical, such as detecting fraudulent transactions. When the cost of building static data structures can be amortized, such as in Louvain or multiple concurrent top-k queries, we build partition-local CSR or CSC data structures, for algorithms that will do multiple passes over the data. These ephemeral CSR data structures are constructed from the current committed state of the graph as live data as a *static view* containing all data that should be visible to the algorithm.

Graph algorithms are implemented using *graph kernels* that abstract how graph data is stored and accessed, and leaving the algorithm developer free to focus on the algorithm's input and output, ephemeral state, and the operations that need to be carried out during the algorithm. This is done by implementing a few algorithm-specific call-back functions, such as initialization of ephemeral data; what needs to be done when a vertex or an edge is visited, etc.

The data abstraction and foundations of main loop constructs for iterative algorithms are implemented in two core framework functions: (1) *OneHopExpansion* for a given part and a *frontier*, i.e., a set of input vertices, visits the neighbors of the vertices in the frontier, and calls the graph kernel's appropriate call-back functions; (2) *IterativeAlgorithm* starting with a given frontier, coordinates execution strategies (sequential, 1D parallel, 2D parallel) of OneHopExpansion for all relevant parts, collects the output frontiers, and continues to iterate level by level, converting current output frontiers to the next level's input frontier, until the stopping condition (another graph kernel call-back function) is satisfied. By taking

into account input from Iterative Algorithm, and the properties of input frontier and data in the partition, OneHopExpansion chooses a traversal mode that is either index-driven, or scan-driven. Both scan and index-driven execution are carefully optimized to read only the data needed for the execution by taking advantage of the PAX columnar data storage, as well as zone-based version vectors, to ensure a transactionally consistent view of the data.

The traversal mode can be further optimized for single-pass path-finding algorithms, such as BFS, by dynamically changing the mode based on the current state of the algorithm, which we call the *hybrid* execution mode (which is the default). Such algorithms generally start with a single *source vertex*, hence execution can start with index-driven mode, as the frontier gets larger, it can switch to scan-driven execution to achieve higher performance, and in later stages when the frontier gets smaller, revert back to index-driven mode.

One of the key call-backs that a graph kernel needs to implement is the visitor call-back. Three variants with different call parameters exist for optimizing data access: (1) target vertex, (2) source and target vertices, (3) source and target vertices and edge weight. Some algorithms, such as BFS (if we are not tracking parents), only require the target vertices, which will be added to output frontier. Other kernels may require both source and target vertices, as well as the weight of the edge between them, requiring a fast join between Rel_E and Rel_Ep.

Graph kernels can have 1D or 2D partitioned ephemeral data, conformal with the current topology relation partitions. For example, the BFS level or the PageRank of a vertex are stored in 1D-partitioned ephemeral data structures. Even though topology uses a 2D partitioning system, most of the current algorithms are parallelized over 1D (for S or O), depending on the write/update-pattern of the algorithm, in order to use the most efficient sequential data structures to store/aggregate ephemeral values for vertices.

8 Query Processing

We launched Neptune Analytics with support for openCypher, a popular declarative graph query language with extensions for invoking graph algorithms. We then followed up that initial launch with *OneGraph*-based support for RDF, allowing customers to load and query RDF data using openCypher. The Neptune Analytics stack is explicitly designed to be query language agnostic. This provides us with a path to onboard additional graph query languages like Gremlin, SPARQL, and GQL in the future.

At its core, openCypher provides an ASCII art syntax for matching patterns against the input graph. On top of these basic graph matching facilities, the language offers constructs that map to Relational Algebra like operators such as projections, filters, aggregations, and different flavors of joins (natural join, left outer join, join exists, etc.) [19]. Its type system comprises standard primitive types (boolean, numerics, strings), graph-specific types (nodes, relationships), as well as composite types such as lists, maps, and paths. Algorithm invocation is supported via the `CALL` clause, which is openCypher's built-in mechanism to invoke stored procedures.

The following is an example of an openCypher query that computes the Jaccard similarity between `Jane Doe` and `John Doe`, extracts all relationships between them, folds the relationships into a list, and reports back the score as well as the relationship list:

```
PREFIX foaf : <http://xmlns.com/foaf/0.1/>
MATCH (jane { foaf::firstName : 'Jane', foaf::lastName : 'Doe' })
MATCH (john { foaf::firstName : 'John', foaf::lastName : 'Doe' })
CALL neptune.algo.jaccardSimilarity(jane, john, {})
YIELD score
OPTIONAL MATCH (jane)-[rel]-(john)
WITH score, collect(rel) AS relationships
RETURN score, relationships
```

In addition to providing a look and feel of the openCypher query language, the example demonstrates two Neptune Analytics specific aspects of openCypher usage: (1) the `PREFIX` keyword introduces a namespace `foaf`[2], which is then used to specify RDF specific node labels (`foaf::Person`) and properties (`foaf::firstName`, `foaf::lastName`); it is a syntactic openCypher extension to ease access of global identifiers (IRIs) originating from RDF data loaded; (2) the query illustrates the seamless integration of algorithm invocation via a `CALL` clause and traditional querying.

Queries run through a holistic query processing pipeline. Processing stages include parsing of the input query, the transformation of the parsed AST into a language-agnostic logical AST, the optimization of this logical AST, the translation of the optimized AST into a physical plan, and the execution of the physical plan. As a notable special case, for *parameterized* openCypher queries – a protocol-level feature that enables the separation of input parameters from a parameterized query template, akin to SQL prepared statements – Neptune Analytics leverages a query plan cache that skips parsing, transformation, and optimization in favor of direct instantiation of pre-computed physical plans, which is crucial to minimize redundant work and accelerate OLTP workloads with high throughput and low latency requirements.

While an in-depth description of the query processing pipeline is beyond the scope of this paper, in the rest of this section we sketch key interfaces between the query layer and the Poseidon backend.

8.1 Statistics and Cardinality Estimation

Starting with a discussion of the optimization layer, Neptune Analytics optimizes queries through a sequence of rewrites. Inspired by algebraic equivalences over a graph algebra [29], these rewrites include general normalizations (e.g., join group flattening), redundant pattern optimization, and algebraic reordering of the logical AST using techniques such as filter pushdown, projection pushdown, and, at its core, join order optimization. Finding a good join order is particularly challenging in graphs. The key challenges are that (C1) from a logical perspective, graph data is fully decomposed (into edges and properties – as opposed to wide relations encountered in Relational Databases that group related attributes together) and the re-composition of this decomposed data often induces a *large number of joins* (with tens of joins per query being the norm rather than the exception); (C2) estimating join cardinalities during the re-composition process demands innovative approaches to statistics that *capture correlations in the decomposed relations*; (C3) power nodes introduce *skew in the data that causes high variance in result cardinality*. To illustrate the latter by example, the computation of the three-hop neighborhood for a person in a social network – a sequence of two join expansions from the starting person – may return very few to millions of results, depending on whether the query starts out from (or, during expansion, encounters) a power

[2] `foaf` is a commonly used namespace for RDF data that provides shared vocabulary to describe persons and their relationships, see http://xmlns.com/foaf/spec/.

node. Another common challenge is that (C4) intermediate cardinalities are often highly *sensitive to the traversal direction* (e.g., an edge such as `isPartOf` may expand one-to-one when traversed forward but one-to-many when traversed backwards). To address all these challenges, the optimizer leverages a combination of schema and instance level statistics exposed by the Poseidon backend.

Schema level statistics. Characteristic sets [20] capture structural patterns in the data – concretely, aggregated information about nodes that share the exact same sets of properties and (in our implementation: outgoing) edge labels. In addition, they also provide frequency distributions for these structures. For instance, a characteristic set may capture the information that there are 1M vertices in the graph that have exactly one `foaf::firstName` property, one `foaf::lastName` property, and, on average, three outgoing `foaf::knows` relationships. As such, characteristic sets provide concise cardinality estimates for more complex join groups and capture correlations between predicates and edge labels in the input graph (challenges C1, C2). The Poseidon backend extends the original notion of characteristic sets [20] by additionally providing (a) min and max frequency distributions, which enables the optimizer to assess and quantify the risk of encountering power nodes during traversals (C3) and (b) HyperLogLog sketches [10] that approximate the number of distinct values for a given property or edge, which helps the optimizer estimate join ratios when estimating join orders that traverse in backwards direction (C4).

Instance level statistics. Characteristic sets capture structural information about nodes, properties, and edge labels, but do not provide any *instance level* information. Real-world queries, however, often reference specific nodes. In such cases, it is crucial to obtain cardinality estimates specific to the given nodes, which allows the planner to disambiguate power nodes from moderate or low fan-out nodes. To support the extraction of such instance specific estimates, APL computes cardinality estimations for (primitive, non-join) access paths via sampling, essentially approximating the output cardinality of an APL expression instead of executing it. Complementing the structural information provided by characteristic sets, instance-level cardinality estimates provide additional information to tackle the challenges of skew (C3) and traversal order (C4) for specific starting points provided in the input query.

8.2 Cost Function and Plan Exploration

The cardinality estimates that are derived from schema and instance level statistics are used as input to the cost function of the join order optimizer, which drives a Selinger-style [30] dynamic plan exploration algorithm with aggressive cost-based pruning. Within this exploration process, the cost function is used to incrementally estimate the execution time of joins while choosing the best available physical join operator implementation based on cost estimates.

The physical join operators themselves are configured using APL expressions, which are partitioning scheme agnostic (cf. Section 6). For access paths that are mapped to relation scans (cf. Section 5), the execution time is an estimate of the time required to scan (relevant parts of) the relations required to execute the access path; for index scans (cf. Section 5), we approximate the execution time by summing up the cost of three operations, namely:

```
(indexLookupTime + adjacencyListScanTime) + projectionTime
```

First, `indexLookupTime` captures the time required to locate, for each incoming value in the input frontier, the adjacency list(s) of the respective value; for the above mentioned case of joining against Rel_E, this component depends on the number of Rel_E partitions; for instance, looking up a value against the S-index or O-index of an NxN partitioned Rel_E relation requires N lookups in total. Thus, while the cost constant for `indexLookupTime` is very low, the cost increases logarithmically with the number of Rel_E partitions in a row (or column) of the topology relation.[3] The second component, `adjacencyListScanTime`, approximates the time required to scan through the adjacency list(s) that were identified through the index lookups and filter out false positives (which may occur, for instance, when the APL request contains additional constraints on the P position). Third, `projectionTime` captures the time required to copy all matching data into the produced output columns; this component primarily depends on the estimated join output cardinality as well as the number of projected positions.

8.3 Query Execution

When it comes to the execution of the physical plan, the Poseidon storage and indexing layer is integrated with a purpose-built query execution engine called *dataflow execution engine* (DFE), which can be characterized as an in-memory, columnar, vectorized execution engine that supports pipelined query execution through a set of RISC-style operators [7, 22]. DFE execution plans are represented as directed acyclic graphs whose nodes represent operators and the directed edges represent the flow of data across them.

DFE operators fall into two categories: those that (A) operate directly against the data served by the Poseidon backend (i.e., operators for scans and lookups against the base relations, bi-directional resolution of terms against the dictionary, and operators for the specific graph algorithms as discussed in Section 7) vs. (B) operators that operate on (intermediate) in-memory results (e.g., selection, filter evaluation, union, or aggregation). The clear separation of operators that access base graph data vs. those that operate over intermediate solutions provides a clean abstraction layer and makes DFE easily re-usable over different storage backends, by just swapping out the category (A) operators. In fact, we are using the DFE engine with a different set of data access operators as the default openCypher execution engine for the original Neptune Database product offering that precedes Neptune Analytics (the original platform emphasises OLTP workloads, scales to graphs much larger than memory, hosts three different query languages, but does not support graph algorithms). Finally, graph algorithm operators support internal parallelism.

The DFE engine can be configured to use either single-threaded execution (implemented via co-routines, using a depth-first execution paradigm) or multi-threaded execution (push-based, with each thread taking care of scheduling the next action for that thread when it blocks needing input chunks or fills up an out-edge buffer). Taken together, these two execution modes capture the spectrum

[3] To minimize that cost constant, we replaced the use of concurrent hash maps with direct access arrays backed by virtual memory allocations. To further reduce cost artifacts and improve space utilization for hyper-sparse graphs, we are in the process of introducing a variant of the secondary indexing mechanisms described above where the adjacency list is only partitioned for high degree vertices.

required for an HTAP engine: single-threaded execution is particularly useful for OLTP workloads with high throughput and low latency requirements (no thread coordination and context switching overhead) and pipelined `LIMIT` queries (due to its depth-first execution that minimizes time to first results), whereas multi-threaded execution is particularly attractive to overlap computation and hide latency when executing compute-intensive, OLAP-style queries.

9 Performance Evaluation

Scan Rates. To quantify the performance of Poseidon we compare both index scans and relation scans. We use the LDBC Social Network Benchmark (SNB) [2] at scale factor 10. We deployed a Poseidon instance with a 16×16 2D grid partitioning for the topology relations. All experiments were run on an Amazon R6i instance. For our queries we chose three different vertex types with a bound P value and fired APL scan requests with different sized frontiers. Each of the three vertex types has a different average degree, thus varying the number of results per vertex. Specifically, we used the statement `Comment.HAS_CREATOR` which always has an out-degree of 1, the statement `Forum.HAS_TAG` which has an average out-degree of 3.5, and lastly the statement `Forum.HAS_MEMBER` which has an average out-degree of 28 members.

Table 4 lists the tuples retrieved by probing the index for different sizes of frontiers (i.e., 100,000 and 500,000 frontier sizes). When the out-degree of the queried statement increases, then we notice an increase of the tuples retrieved up to 5 million tuples per second. This is observed because for each index probe we read more tuples from the adjacency list.

Table 5 lists the tuples read by the relation scan operation for differently sized input frontiers. Although relation scans can achieve maximum memory throughput (more than 500M tuples read per second), different factors come into play. Large-sized frontiers entail greater costs for matching tuples to input vertices (through hash probing of the frontier) and also more tuples are materialized per page scanned. Thus, we can see a drop of tuples read when both frontier and out-degree increase.

Table 4: Index scan rates (tuples read M/sec)

statement	frontier size	time (ms)	tpls (M/s)
`Comment.HAS_CREATOR`	100,000	120.75	0.82
`Comment.HAS_CREATOR`	500,000	529.64	0.94
`Forum.HAS_TAG`	100,000	339.21	1.10
`Forum.HAS_TAG`	500,000	1626.19	1.14
`Forum.HAS_MEMBERS`	100,000	599.20	4.79
`Forum.HAS_MEMBERS`	500,000	2833.18	5.09

Table 5: Relation scan rates (tuples read M/sec)

statement	frontier size	time (ms)	tpls (M/s)
`Comment.HAS_CREATOR`	100,000	1446.2	452.8
`Comment.HAS_CREATOR`	500,000	2116.05	312.5
`Forum.HAS_TAG`	100,000	1645.35	407.2
`Forum.HAS_TAG`	500,000	2766.54	266.5
`Forum.HAS_MEMBERS`	100,000	1779.04	376.6
`Forum.HAS_MEMBERS`	500,000	3258.59	226.3

Table 6: Properties of the test graph datasets.

Dataset	# vertices	# directed edges
com-LiveJournal	3,997,963	69,362,378
com-Orkut	3,072,441	234,370,166
twitter7	41,652,230	1,468,364,884
RMAT Scale-26	67,108,864	2,147,483,648

Graph Algorithms. Figure 3 displays a comparison of index-based, scan-based and hybrid BFS implementation, using three popular datasets from SuiteSparse.[4] The characteristics of the three datasets used in this experiment (and a fourth one, RMAT Scale-26, used in the next experiment) are displayed in Table 6. We carried out these experiments on an R6i instance using the Poseidon storage engine without the higher level components to highlight trade-offs in the actual algorithm execution. As described in Section 3, SPOI tuples are stored in a heap – using scans of the heap in early iterations of algorithms such as BFS reads too much data. Index-driven scans use a secondary S (or O) index to find the SPOI locations for specific S (or O), hence they have additional indirection, but are much more selective. When the frontier is large, such as in levels 3-4 in most power-law graphs, this causes too many unnecessary memory indirections, as seen in the last chart of Figure 3 which presents the breakdown of execution time to levels in 16-thread parallel BFS on the twitter7 dataset. Our hybrid approach switches dynamically between index-driven and scan-driven access to achieve the best performance for all datasets in all thread counts.

Figure 4 displays the performance of the Closeness Centrality implementation. The complete (exact) centrality algorithm [4] is equivalent to running one BFS from each vertex in the graph. Thus, for a graph with n vertices and m edges, its runtime complexity is $O(n \cdot m)$, i.e., for each one of the n vertices, it requires $O(m)$ work for each BFS. An approximated closeness centrality can be implemented by running BFS on randomly selected *source vertices*, say with a cardinality of $n' \ll n$, which achieves significant performance improvements [17]. The number of source vertices (n') is an algorithm parameter that can be changed by the user.

To run the approximate algorithm we leverage CPU-SpMM [27]. Furthermore, to improve performance, CPU-SpMM concurrently runs c concurrent BFSes using vectorized Sparse-Matrix Matrix multiplications, and repeats this process until all sampled vertices are used as a source for a BFS. The algorithm can be viewed as expanding the frontier of multiple BFSes level-by-level with each SpMM multiplication [5, 26, 33]. In the ℓ-th multiplication, the sparse matrix represents the sparsity structure of the graph, and the dense matrix can be viewed as a concatenation of multiple columns, each representing the vertices in the current frontiers of one of the concurrent BFSes in the ℓ-th level. If the number of concurrent BFSes (c) is small (in the extreme it is 1, and the algorithm runs a single BFS), this algorithm will be less work efficient than the naive BFS algorithm. For a graph with a diameter of d, its

[4] SuiteSparse: https://suitesparse-collection-website.herokuapp.com/

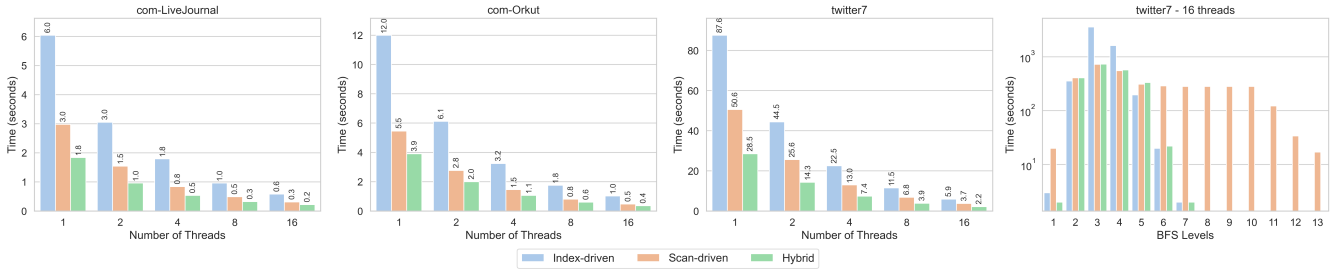


Figure 3: Comparison of index-based, scan-based and hybrid (default) BFS performance.

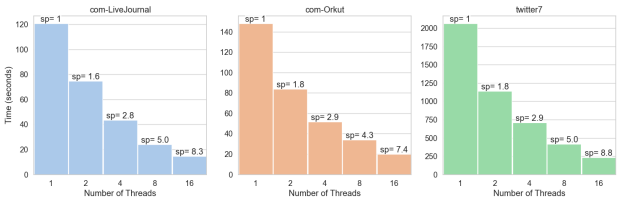


Figure 4: Closeness Centrality Performance on test datasets, using $n' = 2,048$ source vertices.

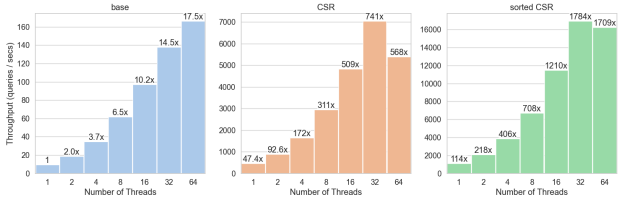


Figure 5: Performance of the EgoNet variants.

runtime is $O(d \cdot m)$ instead of just $O(m)$. However, if $c > d$, which is true in many real-world graphs, this algorithm will be more work efficient. Furthermore, since the access pattern of SpMM is more sequential than BFS, CPU-SpMM [27] runs much faster than previous state-of-the-art CPU implementations. In our experiments we used $c = 2,048$. Our implementation can heuristically choose a smaller value for c (up to 32) when there is not enough memory to run the algorithm with larger c values.

Figure 5 displays the performance of the EgoNet algorithm on RMAT Scale-26. The EgoNet algorithm finds the egocentric network of a given vertex to its *hopCount* neighbors. At each level, the algorithm includes top k maximum (or minimum) weighted neighbors. In this experiment, 128 EgoNet queries run with $k = 100$ and $hopCount = 2$ while varying the number of threads used to concurrently run this experiment. The first plot displays the performance of the base algorithm running against dynamic live data. This would require a two-level BFS-like traversal with filtering using index-driven traversal. As seen in the figure, in this implementation, throughput can be improved up to 17.5× using 32 threads. The second plot displays the performance of the algorithm running on top of unsorted CSR. Use of CSR, in particular, having edge and edge weight stored together in CSR, can improve the performance 47.4× on sequential execution and up to 741× using 32 threads. Since the algorithm searches for top $k = 100$ neighbors, if the CSR is generated by sorting the neighbors by weight (in any order), the algorithm can take advantage of this by looking at either the first or the last k edges in the adjacency list, depending on sorting order. The last plot displays the performance of the algorithm leveraging sorted CSR. As seen in the figure, this gives an additional 2.4× improvement over CSR, hence making improvement over base implementation up to 114× for sequential execution, and up to 1784× improvement by using 32 threads.

10 Conclusion and Future Work

We have described Poseidon, a novel hybrid transactional and analytic engine for graph workloads. The Poseidon engine has been in production since AWS re:Invent 2023, powering the Neptune Analytics service. Neptune Analytics users have created thousands of graphs since launch. Recently, a new group of users is using Neptune Analytics to support their GraphRAG[5] workloads. Overall, we have achieved our initial design objectives, delivering a high performance graph engine to Amazon customers. Users are able to use graph traversals with HPC performance against live data while multi-pass graph analytics transparently use HPC indexing strategies when the cost of building the index is justified. Query workloads automatically switch between relation scans and index scans based on the selectivity, and the query optimizer has access to detailed statistics and a detailed cost model of engine operations.

Decoupling the in-memory indices from the durable storage using a logical log has made it possible for us to continuously optimize the engine design, more than doubling the load performance since the service was announced. We discovered some performance artifacts in production (especially for Rel_E) and have been able to address those concerns by replacing the use of concurrent hash maps with simple arrays backed by virtual memory allocations. Future work will further smooth out performance using a hybrid indexing strategy, which also improves data density for hyper-sparse graphs, replace the remaining concurrent hash map over "O" property values with a thread-safe ordered index; and introduce zone maps, small aggregates, and imprints [32] to further accelerate relation scans and data access.

Acknowledgments

We would like to acknowledge all of the members of the Amazon Neptune team for their contributions to Neptune Analytics. A special thanks to Tengiz Kharatishvili for his insightful contributions on transactions and durability.

[5]https://aws.amazon.com/about-aws/whats-new/2025/03/amazon-bedrock-knowledge-bases-graphrag-generally-available/

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*.

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. 2024. The LDBC Social Network Benchmark. arXiv:2001.02299 [cs.DB] https://arxiv.org/abs/2001.02299

[3] Scott Beamer, Krste Asanović, and David Patterson. 2015. *The GAP benchmark suite.* Technical Report. arXiv preprint arXiv:1508.03619.

[4] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249 arXiv:https://doi.org/10.1080/0022250X.2001.9990249

[5] Aydın Buluç and John R. Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496 – 509. https://doi.org/10.1177/1094342011403516

[6] Ümit V. Çatalyürek, John Feo, Assefaw H. Gebremedhin, Mahantesh Halappanavar, and Alex Pothen. 2012. Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures. *Parallel Comput.* 38, 10-11 (Oct-Nov 2012), 576–594. https://doi.org/10.1016/j.parco.2012.07.001

[7] Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System.. In *VLDB*. 1–10.

[8] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J. Carroll, and Brian McBride. 2014. *RDF 1.1 Concepts and Abstract Syntax.* W3C Recommendation. World Wide Web Consortium. http://www.w3.org/TR/rdf11-concepts/

[9] Martin J. Dürst and Michel Suignard. 2005. *Internationalized Resource Identifiers (IRIs).* RFC 3987, Proposed Standard. IETF. https://www.rfc-editor.org/rfc/rfc3987

[10] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete mathematics & theoretical computer science* Proceedings (2007).

[11] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying.. In *EDBT*. 520–523.

[12] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying. In *Proceedings of the 21st International Conference on Extending Database Technology (EDBT)*. https://doi.org/10.5441/002/edbt.2018.62

[13] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language.* W3C Recommendation. World Wide Web Consortium. https://www.w3.org/TR/sparql11-query/

[14] Olaf Hartig, Pierre-Antoine Champin, Gregg Kellogg, and Andy Seaborne. 2021. *RDF-star and SPARQL-star.* W3C Community Group Report. World Wide Web Consortium. https://www.w3.org/2021/12/rdf-star.html

[15] Ora Lassila, Michael Schmidt, Olaf Hartig, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Carlos Manuel López Enríquez, Ronak Sharda, and Bryan B. Thompson. 2023. The OneGraph Vision: Challenges of Breaking the Graph Model Lock-In. *Semantic Web* 14, 1 (2023), 125–134. https://doi.org/10.3233/SW-223273

[16] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 1 (2007), 5–20.

[17] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2009.5161100

[18] Yury A. Malkov and Dmitry A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *CoRR* abs/1603.09320 (2016). arXiv:1603.09320 http://arxiv.org/abs/1603.09320

[19] József Marton, Gábor Szárnyas, and Dániel Varró. 2017. Formalising openCypher graph queries in relational algebra. In *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings 21*. Springer, 182–196.

[20] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 984–994.

[21] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[22] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659.

[23] Ian Robinson, Jim Webber, and Emil Eifrém. 2015. *Graph Databases: New Opportunities for Connected Data* (2nd ed.). O'Reilly Media, Inc.

[24] Marko A Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*. 1–10. https://doi.org/10.1145/2815072.2815073

[25] Marko A Rodriguez and Peter Neubauer. 2010. Constructions from Dots and Lines. *Bulletin of the American Society for Information Science and Technology* 36, 6 (2010), 35–41. https://doi.org/10.1002/bult.2010.1720360610

[26] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. 2014. Hardware/Software Vectorization for Closeness Centrality on Multi-/Many-Core Architectures. In *28th International Parallel and Distributed Processing Symposium Workshops, Workshop on Multithreaded Architectures and Applications (MTAAP)*. https://doi.org/10.1109/IPDPSW.2014.156

[27] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. 2015. Regularizing Graph Centrality Computations. *J. Parallel and Distrib. Comput.* 76 (Feb 2015), 106–119. http://www.sciencedirect.com/science/article/pii/S0743731514001282

[28] Michael Schmidt, Brad Bebee, Willem Broekema, Mohamed Elzarei, Carlos Manuel López Enríquez, Marcin Neyman, Florian Schmedding, Andreas Steigmiller, Bryan B. Thompson, Geo Varkey, Gregory Todd Williams, and Amanda Xiang. 2024. openCypher over RDF: Connecting Two Worlds. In *Proceedings of the ISWC 2024 Posters, Demos and Industry Tracks: From Novel Ideas to Industrial Practice co-located with 23nd International Semantic Web Conference (ISWC 2024) (CEUR Workshop Proceedings)*, Vol. 3828. CEUR-WS.org.

[29] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *Proceedings of the 13th international conference on database theory*. 4–33.

[30] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[31] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, Shenzen, 135–146.

[32] Lefteris Sidirourgos and Martin Kersten. 2013. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 893–904.

[33] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The more the merrier: efficient multi-source graph traversal. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 449–460. https://doi.org/10.14778/2735496.2735507

[34] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. 2012. Heuristics-based query optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology* (Berlin, Germany). Association for Computing Machinery, New York, NY, USA, 324–335.

[35] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan W. Berry, and Ümit V. Çatalyürek. 2022. *PGAbB: A Block-Based Graph Processing Framework for Heterogeneous Platforms.* Technical Report arXiv:2209.04541. ArXiv. https://arxiv.org/abs/2209.04541

[36] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Ümit V. Çatalyürek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of SC2005 High Performance Computing, Networking, and Storage Conference*. https://doi.org/10.1109/SC.2005.4 Gordon Bell Finalist.