# Literature Survey of Graph Databases

Bryan Thompson, SYSTAP

23 January 2013

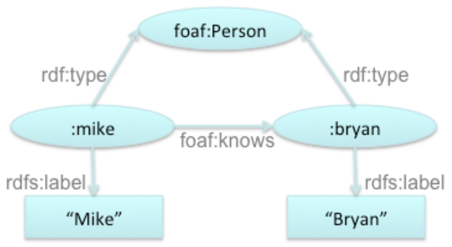
## Introduction

The Resource Description Framework (RDF) (Lasilla, 1998) and the SPARQL query language (Prud'Hommeaux, 2008) provide standards for the interchange, query and update of graph data in database systems. RDF was originally developed as a metadata model for web resources, hence the *Resource* Description Framework.

The RDF data model describes a graph in terms of its edges. An edge is a *triple*:

<Subject, Predicate, Object>

where the Subject is the source vertex;
where the Predicate is the *link type*; and
where the Object is the target vertex.



The vertices in RDF are URIs, Literals, or blank nodes. The use of URI as resource identifiers makes it possible to share ontologies (aka schema) and data sets using common identifiers for well-known concepts or entities. Literals may be data-typed using the XML Schema Datatypes (XSD) (Biron, 2004) standard. Blank nodes represent anonymous variables, but are often misused as unique resource identifiers. Subjects must be Resources (URIs or blank nodes). Predicates must be URIs. Objects may be either Resources (in which case the triple represents a *link*) or Literals (in which case the triple represents a named attribute value).

SPARQL provides a high-level declarative query language based on graph patterns. SPARQL 1.1 adds sub-selects, aggregation, a graph update language, and a language for federated graph query. A variety of software tools provide the ability to query non-graph structured data sources using SPARQL queries, thus supporting federation of heterogeneous data sets and heterogeneous kinds of databases.

The SPARQL language is similar in many respects to the Structured Query Language (SQL), but builds on graph pattern matching as its core operation. For example, the following query finds all friends of friends that are not direct friends and that share at three or more indirect connections.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?x, ?z, (COUNT(?y) as ?connectionCount)
WHERE {
   ?x foaf:knows ?y .
   ?y foaf:knows ?z .
   FILTER NOT EXISTS { ?x foaf:knows ?z }
   FILTER ( ?x != ?y )
}
GROUP BY ?x ?z
HAVING (?connectionCount >= 3)
```

The basic unit in a SPARQL query is a triple pattern, which is simply a triple in which some positions may be variables. Shared variables establish joins. Query evaluation identifies solutions that match the individual triple patterns and obey the constraints imposed by shared variables and FILTERS. Optional joins (similar to left outer joins in SQL) may be used to pull back data that is desired, but where the solution should succeed even if the optional join fails.

Complex SPARQL queries are common. We routinely observe queries with 50 to 100 joins. Fast evaluation of such queries depends on rapidly identifying the most selective triple patterns and managing the intermediate cardinality. With good join orders, complex queries typically have sub-second execution times. Traditional cost based query optimizers are at a disadvantage due to the large state space, leading to the development of new query optimization techniques.

Unselective queries are often used in aggregations and data analytics. Efficient evaluation of unselective queries relies on not only good join orders, but also access paths that are efficient for large key-range scans. For unselective queries, the cost of hidden data skew and correlations between the query and the data can dramatically increase the running time of the query. Runtime optimization techniques (Kader, 2009; Kotoulas, 2012) have been developed for such analytic workloads based on cutoff sampling of join paths.
The following is an example of an auto-generated analytic query that performs a rollup of votes by Democrats on various bills against the Govtrack (http://www.govtrack.us/) data set. Significantly more complex analytic queries are routine.

```
PREFIX p1: <http://www.rdfabout.com/rdf/schema/usgovt/>
PREFIX p2: <http://www.rdfabout.com/rdf/schema/vote/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT (SAMPLE(?_var9) AS ?_var1) ?_var2 ?_var3
WITH {
        SELECT DISTINCT ?_var3
        WHERE {
                ?_var3 rdf:type <http://www.rdfabout.com/rdf/schema/politico/Politician>.
                ?_var3 <http://www.rdfabout.com/rdf/schema/politico/hasRole> ?_var6.
                ?_var6 <http://www.rdfabout.com/rdf/schema/politico/party> "Democrat".
        }
} AS %_set1
WHERE {
        INCLUDE %_set1 .
        OPTIONAL {
                ?_var3 p1:name ?_var9
        }.
        OPTIONAL {
                ?_var10 p2:votedBy ?_var3.
                ?_var10 rdfs:label ?_var2.
        }
}
GROUP BY ?_var2 ?_var3
```

While SPARQL is a powerful language for graph query and is well suited for expressing sophisticated queries and complex aggregations, both SPARQL and the existing SPARQL implementations are unable to adequately express and compute variety of interesting graph data mining operations. However, extensions of

SPARQL have been developed for spatial data (geoSPARQL) (OGC, 2011) and stream data (Della Valle, 2009; Bolles, 2008; Anicic, 2011). We consider it very likely that an extension could be developed that would support the expression of graph mining algorithms and their efficient evaluation on a variety of compute clusters.

## Linked Data

Linked Data is the concept that a URI identifies a data set *describing* that resource. If you *de-reference* the URI (using an HTTP GET), you obtain the corresponding data set. This provides a simple, but robust basis for federated query against the open web. The linked data interface is much simpler than a SPARQL endpoint, but this simple robust interface makes it possible to expose resources to the web in an open and scalable fashion.

The Linked Data Cloud is a meta-collection of data sets the share some ontologies and cross-link some concepts and instances. The linked data cloud is published both as complete data sets by various web sites (such as DBpedia (Auer, 2007) or geonames (*http://geonames. org*) and as *linked data resources* (such as a URI for a specific city, politician, actor, gene, or protein). SPARQL end points also exist for some of these data sets, such as DBpedia. However, these public endpoints are offered on an "as-is" basis. As such, they are often unreliable and/or have resource limits that make them unsuitable for building robust federated query applications. Also, these data are independently maintained and not truly semantically integrated, as discussed below.

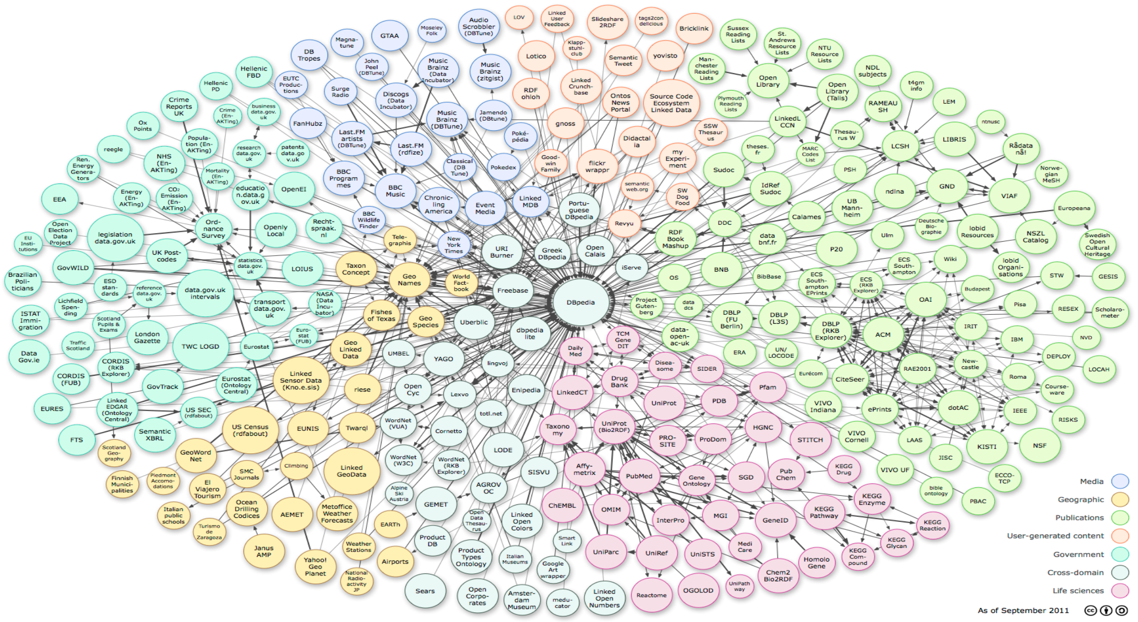


Figure 1 - The Linked Open Data Cloud

## Graph databases (blueprints)

The blueprints API (https://github.com/tinkerpop/blueprints/wiki) supports an implied data model that is similar to RDF. The basic distinction is that the blueprints API specifies a navigational and procedural interface to graph data while SPARQL specifies a declaration pattern machine interface to graph data. The main differences between the blueprints model and RDF are:

- The blueprints data model allows property sets to be attached to both vertices and links. RDF statements directly model property sets for vertices. RDF reification provides a mechanism for modeling link attributes for the purposes of data interchange and query. A common point of confusion, and one that has been poorly handled by the RDF community, is how to handle link attributes efficiently. However, the logical / physical schema distinction means that the database vendor can make intelligence choices when indexing and querying link attributes. For example, both the bigdata and virtuoso platforms support inlining of link attributes into the statement indices so they are tightly clustered in the indices rather than storing them in a fully normal form. The diplodocus platform also supports efficient handling of link attributes through its sub-graph templating model (both links and link attributes can be tightly clustered with the property sets for the relevant vertices). The efficient handling of link attributes is thus properly see as a database design decision, not an intrinsic strength or weakness of a data model or a query language.
- Blueprints permits simple attributes as vertex identifiers. However, the corollary is that you cannot share data or ontologies since there is no basis for publishing global identifiers.
- The blueprints API encourage programmers to write procedural code that uses a navigational pattern against the graph data model. Someone has quipped that this "is the full employment act for programmers". Programmers must optimize procedural code by hand for each query. This procedural / navigational API effectively prevents the database from optimizing access patterns for complex queries. RDF databases use a high-level declaration query language based on pattern matching and use query optimizers to deduce efficient query plans, including plans that are sensitive to data skew (e.g., using robust online sampling of join paths).
- Graph databases often (but not always) provide property set recovery in a single index lookup. This is similar to the single index lookup guarantee of a key-value store for a property set. RDF databases often (but not always) encode the attributes into dictionaries and must materialize attribute values in order to materialize a property set. As a consequence, RDF databases tend to be significantly faster for complex pattern matching queries but slower for recovering a single property set. However, this distinction is become blurred as indexing schemes evolve and RDF databases begin to optimize for linked data query patterns. For example, bigdata maintains a cache for property set descriptions so it has an amortized lookup cost of one index lookup while diplodocus directly stores the property set and link sets for each vertex in a scalable, page-oriented data structure but also supports efficient high-level declarative queries evaluation based on pattern matching.
- Relational databases commonly use application-defined indices based on combinations of column appearing in a relation. Blueprints style databases

tend to combine navigation patterns (pointer chasing against disk) and application-defined indices on attribute of object collections (lookup vertices collected by a link set using a specific index). One consequence of the schema-late or schema-never perspective of RDF databases is that applications rarely concern themselves with the choice of indices – in fact, the SPARQL query language completely lacks the concept of an index. When custom indices are used in RDF databases, they are generally extensions for new data types (spatial) or history (temporal tables) and they are created by a DBA, not a programming writing application level queries. Occasionally application-defined indices maybe used to accelerate certain queries. One project using map/reduce to answer SPARQL queries takes this approach to the extreme and can *only* answer queries for which perfect indices have been pre-computed. This is essentially a query specific index.

- Graph databases often provide libraries for graph primitive operations such as breadth-first traversal or shortest-path. However, these primitives are generally extremely inefficient when compared to main memory distributed graph mining platforms (Spark's Bagel, Apache Giraph, etc.)
- The Linked Data Benchmark Council (LDBC) is a new effort to develop benchmarks for RDF databases and blueprints style graph databases, including both SPARQL query and graph algorithms. It may be that some convergence in the market will result from this effort as use cases from different communities are captured within common benchmarks.

## Challenges

The Linked Open Data Cloud is an impressive example of what can be achieved with standards for graph data interchange and simple access protocols. However, there are significant problems introduced by the success of RDF, SPARQL, and linked data. First, these data sets are not truly semantically integrated – instead they tend to share some schemas and provide some level of cross-linking. As more and more graph data sets are published (or shared privately by corporations), it becomes increasingly important to find ways to *automatically* semantically align and mine the relationships in these data. Otherwise we will find ourselves with too much data and too little information that we can actually use. Second, as more and more data sets are combined, it becomes increasingly impossible for even expert users to write queries that properly integrate all of the available data that bears on their problem. This leads quickly to published and semantically disclosed information that nevertheless cannot be adequately leveraged.

Some of the central research challenges for graph query are:

- Schema alignment. Shared ontologies, such as the Dublin Core (Weibel, 1998), GeoNames, or the UMLS (Lindberg, 1993; McCray, 1995) provide a rough and ready entrance into new data sets but do not address the real variation in the application of those ontologies to the data and do not help when new data sets must be combined on an ad-hoc basis. Even the Unified Medical Language System (UMLS), a gold standard for curated metadata, has an inter-rater reliability of 50% among trained professionals tagging medical abstracts. Thus shared ontologies are only a partial solution, and schema alignment remains a problem both within and across data sets. This problem is only exacerbated as the numbers of data sets to be considered grows.

- Schema agnostic query. When only a few data sets are combined, it is possible for users to understand their schema and write structured queries. However, as the number of data sets grow, this becomes first difficult and then impossible. Instead, schema agnostic query pursues techniques that allow people to ask natural language questions and have machines identify possibly interesting patterns in the data, often based on instance level schema. There are also challenges in how to represent the results and provide people with guidance on how to refine or expand these initial results.
- Schema alignment and (by extension) schema agnostic query is a hard problem, and perhaps a problem that can only be "solved" by experts sitting down and designing crosswalks between data sets. Even then, the interpretation of the data requires expertise. Graph mining may provide another approach to these questions. Rather than attempting to reconcile the schema and answer crisp queries against the federated and aligned data sets, applications could apply data mining algorithms that are robust in the face of noisy data and treat schema alignment as another source of noise in the data.
- Queries against the open web. Here, the goal is to answer questions against the open web, using only your laptop and a connection to the Internet. Rather than relying on Internet aggregators such as Google, queries are directly evaluated against the open web using linked data principles to obtain sub-graphs for entities and concepts of interest. URDF (Theobald, 2012) extends this to support reasoning about the reliability of the information obtained from open web queries based on a system of soft rules. This allows it to consider conflicting conclusions, incomplete evidence, and unreliable assumptions. In combination with computational models of critical thinking (The Recognition / Metacognition architecture – Cohen and Thompson, 2001; Cohen and Thompson, 2005), you might assign a budget and a time limit to the query engine. The engine would then iteratively deepen the model, treating conflicting evidence as an indication of structural uncertainty in the model, finding support to back up assumptions through directed search for both supporting and contradicting evidence, and incrementally improving the confidence in its solutions.
- Finding interesting patterns or predictions in graph data (graph data mining). Data mining seeks to discover interesting patterns or fit models to data. One of recent developments in graph data mining is the *vertex programming* abstraction. This abstraction provides users with a narrow procedural API in which they write a program to be executed at each vertex in a graph. These programs can either send or receive messages from other vertices or edges, combining the data in a variety of ways as messages are consumed and generated. A wide array of data mining techniques can be implemented using vertex programs, but there is some doubt (see Dr. Yan's literature review) that vertex programs can handle the rich variety of graph mining algorithms. Research challenges exist in efficient scaling of graph data mining to large clusters, parallelization of graph data mining on many core hardware,

appropriate abstractions for graph data mining APIs that simplify the implementation of a wide variety of data mining algorithms while allowing the underlying framework to efficiently parallelize the work, and integrating graph data mining with rich graph models, including typed links, link weights, and attribute values, and the efficient use of indices to support graph mining operations.

## Survey of Graph Database Platforms

### Interesting graph database research platforms (non-clustered)

#### *RDF3X (Max-Plank Institute)*

RDF3X (Neumann, 2010) departs from other triple store designs by extensive use of *merge joins* (Wolf, 1990) and *sideways information passing* (SIP) (Bancilhon, 1985). In order to maximize the opportunity for merge joins, RDF3X carries 12 indices over the edges in the graph – this compares with 3 indices for a standard triple store or 6 indices for a quad store. Further, RDF3X issues access path scans in parallel on each triple pattern in the query. In order to reduce the required effort, sideways information passing between joins having correlated index orders is used to skip past keys that have been proven absent by another access path. This sideways information passing technique depends on the use of a fixed with integer representation for all lexical forms, including data typed literals. Because the identifiers are known to be dense and in ascending order on an access path scan, gaps observed in a scan identify key ranges that can be skipped on correlated access paths. RDF3X is one of the fastest RDF database platforms in benchmarks due to the use of sideways information passing and merge joins. However, it is unable to handle large query plans due to the explosion in the size of the state space in the cost-based query optimizer. Some research (Huang, Abadi, and Ren, 2011) has explored the use of standalone RDF3X databases instances on the nodes of a compute cluster – this is reviewed below.

#### *Diplodocus (eXascale Infolab)*

While there is not yet a clustered version diplodocus [the authors plan to publish on a clustered version shortly as of 1 Dec 2012 – private communication] and while queries must be hand compiled, the architecture of this database is quite interesting. Unlike most graph databases, diplodocus indexes sub-graphs (also known as “molecules” or “clusters”). Diplodocus claims speedups of 30x (for LUBM) to 350x (for aggregation queries) over existing solutions.

Diplodocus uses a declarative storage pattern based on **templates**. A template is an abstraction of a triple pattern. Rather than being specified for a concrete subject or object, a template pattern matches all uses of some predicate where the subject-type and the object-type are consistent with the pattern.

Template:: Subject-Type, Predicate, Object-Type => TemplateID

External RDF Values are mapped onto internal identifiers using a trie. Three data structures are then hung from a central hash index whose key is an internal identifier. These structures are:

- **Clusters**
- **Cluster-Lists**

In addition, diplodocus maintains for each template:

- **Literal-Lists** (these are also referred to as template-lists in the paper)

Templates serve to group the triples (in clusters), the cluster-lists, and the literal-lists by the different "senses" of the predicate. For example, using the LUBM data, the following templates might be defined (among others):

```
Template:: (FullProfessor, teacherOf, Course)
Template:: (FullProfessor, teacherOf, GraduateCourse)
Template:: (AssociateProfessor, teacherOf, Course)
Template:: (AssociateProfessor, teacherOf, GraduateCourse)
```

These templates would cause the different senses of the *teacherOf* relation to be grouped separately, potentially providing an efficiency gain when a specific sense is relevant to the query.

The **clusters** are subgraphs. The clusters are stored using an encoding that indexes into the triples based on the TemplateID. This groups the triples by the *sense* of the template and allows the predicate to be elided within the triples themselves. A jump table is provided into the triples for a given template within each cluster.

Based on private communication with the authors, it appears that diplodocus only indexes the attributes and forward links within a molecule (i.e., it does not index the reverse links). Thus, as configured for the experiments with a 1-hop (forward) neighborhood, it is simply capturing the SPO index rather than "materializing" joins. For the default behavior (1-hop forward subgraph clusters), the jump table into the cluster is similar to a P:**O** column projection and S is constant for the cluster.

The authors have clarified (private communication) that the reverse dictionary is not co-located with the molecule, thus requiring random IOs for RDF Value materialization. Since diplodocus does not take advantage of inlining XSD datatype values, this could substantially hurt performance when materializing RDF Values or applying FILTERs (except when the query can take advantage of the literal-lists – see below).

The **cluster-lists** are an ordered list of the clusters in which an internal value appears as an object in a triple matching a specific template. The cluster-list is simply a column projection of the cluster identifiers. These ordered lists are used to filter out clusters in which a join cannot succeed based on the intersection of these the cluster-lists.

The performance speed up for diplodocus for non-aggregation queries appears to come from entirely from the cluster-lists. Given the regularity of the data set over which diplodocus has been tested (LUBM), this is similar to an OP:**<u>S</u>** index, where **<u>S</u>** is the ordered list of clusters (aka cluster-list) in which that O appears as the object of a triple whose predicate is P. However, it is more accurate to consider this an OT:**<u>S</u>** index, where T is the template identifier, since templates differentiate among multiple senses of a given predicate.

Merge joins on the cluster-lists can be used to rapidly subset the clusters that must be examined. RDF3x maintains an OPS index and, presumably, should be able to apply it to efficiently filter out clusters that cannot join (that is, merge-joins on OPS could establish a filter that would exclude certain S ranges on the SPO index). However, it lacks the sub-division of the predicate by *sense* that is provided by the template mechanism in diplodocus.

The **literal-lists** are organized by template. For each template, the literal-list is a projection of a names column (the lexical form of the RDF values, in lexical order) and a cluster identifiers column (the identifiers of the clusters in which those RDF values appear). This amounts to a P:**<u>OS</u>** index with *materialized* Os (again, more correctly, a T:**<u>OS</u>** index). The materialized O column projection provides the ordered list of external values. The S column projection provides the correlated list of the clusters in which a given PO (more correctly, TO) combination appears.

Literal-lists are used to accelerate key-range scans and some forms of aggregation queries.

- For a key-range scan, the relevant section of the names column is rapidly identified. The RDF Values and cluster IDs that satisfy the key-range may then be read off from the respective columns.
- If all literals matching the template are desired, the query result may be directly read off of the names column.
- If the intersection of two or more template patterns is known to subsume the query, then an ordered merge-join on the clusters columns for those templates will provide a list of those clusters in which the query could be satisfied.

To the best of our knowledge, no other graph database architecture maintains ordered projections of the external forms of the literals. Thus, all other database must do more work to materialize literals for queries such as "give me all graduate students". For diplodocus, this query is answered by reading off the names column for the appropriate template.

Databases that support inlining, such as Virtuoso or bigdata, can also evaluate key-range queries efficiently (at least for numeric values). For bigdata, assuming that the predicate was specified (as it must be to match a template) the POS index would be used to support key-range scans against O for a constant P. If the predicate is not specified, then key-range scans are evaluated against OSP.

The key innovation of diplodocus appears to be capturing different *senses* of predicate as templates. For predicates where the subject-type and the object-type are highly correlated (e.g., Person foaf:knows Person) this classification into templates will not add any power since it does not create distinctions beyond the predicate itself. However, for predicates where the subject-type and the object-type are not strongly correlated, templates can add substantial power by creating interesting groupings of the data. Essentially, templates *disambiguate* the different role player types for predicates.

Diplodocus generalizes the concept of the cluster to allow more than a one-hop forward neighborhood as part of its declarative storage pattern. No results have been presented for such configurations.

### GraphChi (IO Efficient Graph Mining)

GraphChi (Kyrola, 2012) is an IO efficient graph mining system that is also designed to accept topology updates based on a Parallel Sliding Window (PSW) algorithm. Each iteration over the graph requires P^2 sequential reads and P^2 sequential writes. Because all IO is sequential, GraphChi may be used with traditional disk or SSD. The system is not designed to answer ad-hoc queries and is not a database in any traditional sense – the isolation semantics of GraphChi are entirely related to the Asynchronous Parallel (ASP) versus Bulk Synchronous Parallel (BSP) processing paradigms. GraphChi does not support either vertex attributes or link attributes.

The basic data layout for GraphChi is a storage model that is key-range partitioned by the link target (O) and then stores the links in sorted order (SO). This design was chosen to permit IO efficient vertex programs where the graph was larger than main memory. While GraphChi does not support cluster-based process, the approach could be extended to a compute cluster. Because of the IO efficient design, the approach is of interest for out-of-core processing in hybrid CPU/GPU architectures.

GraphChi operates by applying a utility program to split a data set into P partitions, where the user chooses the value of P with the intention that a single partition will fit entirely into main memory. The edges are assigned to partitions in order to create partitions with an equal #of edges – this provides load balancing and compensates for data skew in the graph (high cardinality vertices).

GraphChi reads one partition of P (called the memory partition) into core. This provides the in-edges for all vertices in that partition. Because the partitions are divided into target vertex key-ranges, and because partitions are internally ordered by the source vertex, out-edges for those vertices are guaranteed to lie in a contiguous range of the remaining P-1 partitions. Those key-ranges may vary in their size since the #of out-edges for a vertex is not a constant. Thus, for the current memory partition, GraphChi performs 1 full partition scan plus P-1 partial partition scans.

In addition to the edges (network structure), GraphChi maintains the transient graph computation state for each edge and vertex. The edge and vertex each have a user assignable label consisting of some (user-defined) fixed-length data type. The vertices also have a flag indicating whether or not they are scheduled in a given iteration. The edge state is presumably read with the out-edges, though perhaps from a separate file (the paper does not specify this). The vertex state is presumably read from a separate file (again, this is not specified). After the update() function has been applied to each vertex in the current memory partition, the transient graph state is written back out to the disk. This provides one more dimension of graph computation state that is persisted on disk, presumably in a column paired with the vertex state.

The consistency guarantee provided by GraphChi is that given a vertex A whose identifier is ordered before a vertex B, A will execute *before* B. This provides a serializable (and hence deterministic) evaluation guarantee. The vertices within a given partition are evaluated in parallel. If there are vertices within a given partition whose in-edges are also out-edges lying within the same partition, then additional steps must be taken to ensure that serializability is retained. This is easily decided by observing whether the target of the out-edge lies inside of the current partition. If the natural order of the vertices does not form a cycle, then data parallel evaluation is still allowed. If a cycle would be formed, then part of the vertex evaluation schedule must be serialized to maintain the serializable execution semantics. GraphChi permits the user to disable this serializable guarantee.

## Extensions to GraphChi

We discuss several potential extensions to the Parallel Sliding Window (PSW) algorithm used by GraphChi, including (1) attributed graphs; (2) column compression; (3) integration with a graph database; (4) GPU accelerated processing; and (5) multi-node extensions.

### *The RDF Graph, Reification, and Link Attributes*

We propose to expose the full RDF graph and data model. If we assuming inlining of attribute values (possibly excluding URIs and Strings), then attribute values can be interpreted directly without resolving them against a dictionary. This makes it possible to avoid all random IO when mining the graph. This approach is possible with a number of platforms, including Virtuoso, bigdata, Accumulo, YAR2, etc.

Link attributes are represented in RDF interchange syntax through RDF Reification. Reification introduces an anonymous subject that stands for the link, describes a model of the link, and then attaches other assertions to that anonymous subject that are the link attributes. For example, the following provides (a) a ground triple; (b) a reified model of that ground triple; (c) a statement about that ground triple made using the reified model (a link attribute); and (c) a statement providing the confidence in that ground triple made using the reified model (a link weight, that is, a link attribute whose value is a weight).

```
(:mike :worksFor :systap)               // ground triple
(_:s1 rdf:hasSubject :mike)             // 4 stmts in reified model.
(_:s1 rdf:hasPredicate :worksFor)
(_:s1 rdf:hasObject :systap)
(_:s1 rdf:type rdf:Statement)
(_:s1 dc:source http://www.systap.com) // link attribute
(_:s1 :confidence 0.8)                  // link weight
```

Please note that RDF Reification DOES NOT determine how the link attributes are represented in the database. This is decided by the physical schema of the database. This choice is entirely up to the database vendor - there are many possibilities.

The bigdata platform co-locates link attributes with the ground triples that they describe. Co-location is achieved by inlining the ground statement into the representation of the link attribute for that statement. For example, the following two assertions are (a) a ground triple; (b) a statement about that statement (a link attribute); and (c) the confidence in that ground triple (a link weight).

```
(:mike  :worksFor :systap)                // ground triple
((:mike :worksFor :systap)
      dc:source http://www.systap.com)    // link attribute
((:mike :worksFor :systap)
        :confidence 0.8)                  // link weight
```

These statements about statements are stored efficiently in the various graph indices. A variety of compression schemes are used to make the storage model efficient, even when there are many attributes for the same link. While the compression is transparent to most applications, low-level applications (including user kernels for graph mining) would need to be aware of the means by which link attributes were represented.

In order to have the statements about statements co-located with the original statement, the inline representation depends on the natural order of the index in which the edges are stored. For the SPO index, the link attribute is essentially represented as ((S,P,O),L,A), where L is the property type for the link attribute and A is the value for the link attribute. The nested (S,P,O) is detected by a postfix marker in the encoding in order to preserve co-location. The original components of the edge can be decomposed trivially to recover the ground statement. Note that this postfix marker allows statements about statements where the ground statement plays the role of the subject or object - this is more general than link attributes.

### *Partitioning for Graph Mining using Parallel Sliding Windows (PSW)*

The RDF Graph is partitioned into P major partitions (called `O` partitions) and P minor partitions (called `S` partitions). The `O` partition boundaries are chosen such that each `O` partition has a balanced number of edges ( `in-edges + out-edges` ). This is done both to control the memory demand associated with a major partition and to balance the work associated with each major partition (assuming that the #of edges is a predictor of the amount of work to be performed).

Each major (O) partition is divided into an equal number of minor (S) partitions. The edges within the `O` partitions are in `SPO` order. The PSW algorithm critically depends on the major and minor partitions having the same boundaries, with the major partitions organized by `O` (the target vertex) and the minor partitions organized by `S` (the source vertex). This layout makes it possible to efficiently represent the graph *without redundancy* and to efficiently assembly the 1-hop neighborhood of all vertices in a major partition. The lack of redundancy and the IO efficiency are critical aspects of the algorithm and allow updates to be written back to the disk in an IO efficient manner.

The partition boundaries are determined by the first vertex to enter into each partition. Thus, each partition has some non-overlapping key-range of `O` vertices (for a major partition) or `S` vertices (for a minor partition).

### *Graph Computation State*

Graph computation *labels* vertices and/or edges with values. Those values are initialized on the first pass of the algorithm and then updated on subsequent passes. The label may be any single XSD datatype (this is not permitted by GraphChi because it relies on a fixed stride) or "ANY" if it the label will be a vertex identifier. If the label has a fixed datatype, then strongly typed columns may be used. The datatype for the vertex state and the edge state are independent. The choice of the datatype depends on the graph mining algorithm. Integer values are used for the vertices in this example in keeping with the GraphChi publication. However, we would recommend using variable length values that support inlining to minimize random IO against a dictionary.

When the graph is partitioned, the first vertex to enter into each partition in recorded into a table along with various metadata such as the fanIn (#of in-edges) and fan-out (#of out-edges) for each major partition.

| partitionNum | firstVertex |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 2 | 5 |

The per-edge data includes the edges and the edgeState for each edge in each OS segment. (See the table below, which is based on the GraphChi paper. Note that "L" is used to indicate the link type and is a constant in this example. The GraphChi paper did not consider link types, vertex attributes, or link attributes.) The edges appear in SPO order in the edge partitions. The minor partition boundaries are marked as well - in the picture they are artificially aligned across the major partitions in order to emphasize the horizontal stripes formed by the minor partitions. If the graph possessed vertex attributes or link attributes, then they would appear as additional rows in each of the major edge partitions.

The edge state is stored as a column projection per OS segment. By separating the data for the edges from the data for the edge state, we are able to capture the edges in key order on the disk and record the edge state in a column segment. This allows us to use compression for both the edges and the edge state. Further, we only need to rewrite the edge state (and not the edges), thus reducing the IO costs associated with the scatter phase of the algorithm.

```
 partition[0] [-,3) | partition[1] [3,5) | partition[2] [5,-) |
--------------------+--------------------+--------------------+
 S P O    edgeState | S P O    edgeState | S P O    edgeState |
--------------------+--------------------+--------------------+ +----------
 1 L 2       0.3    | 1 L 3       0.4    | 2 L 5       0.6    | | out-edges
                    | 2 L 3       0.3    |                    | | (horz. stripe)
--------------------|--------------------|--------------------| +----------
 3 L 2       0.2    | 3 L 4       0.8    | 3 L 5       0.9    | | out-edges
 4 L 1       1.4    |                    | 3 L 6       1.2    | | (horz. stripe)
                    |                    | 4 L 5       0.3    | |
--------------------|--------------------|--------------------| +----------
 5 L 1       0.5    | 5 L 3       0.2    | 5 L 6       1.1    | | out-edges
 5 L 2       0.6    | 6 L 4       1.9    |                    | | (horz. stripe)
 6 L 1       0.6    |                    |                    | |
--------------------+--------------------+--------------------| +----------
----- in-edges -----+---- in-edges ------+----- in-edges -----+
 (vertical stripe)  | (vertical stripe)  | (vertical stripe)  |
```

For major partition `i`, `OS[ij]` is the *vertical stripe* having all *in-edges* for O vertices in the key-range for that major partition. Likewise, `OS[ji]` is the *horizontal stripe* (across the major partitions) having all *out-edges* for major partition `i`. Thus, reading across the horizontal stripe for the first minor partition (`OS[ij]`) we get:

```
S P O  S P O  S P O
-----  -----  -----
1 L 2  1 L 3
       2 L 3  2 L 5
```

The `O` on the `in-edge` (vertical stripe) joins with the `S` on the `out-edge` (horizontal stripe). The horizontal stripe for major partition `i` ( `OS[ij]` ) (as pictured immediately above) therefore describes the one-hop neighborhood of the vertices `1` and `2` ( as pictured immediately below):

```
IN    OUT
---------
    1 L 3
1 L 2        // Note: as in-edge (vertical stripe).
    2 L 1    // Note: as out-edge (horizontal stripe).
    2 L 3
    2 L 5
```

Note that cycles within a major partition ONLY exist for `OS[ii]` . In the example above, that is `1 L 2` for partition ZERO (0), `3 L 4` for partition ONE (1), and `5 L 6` for partition TWO (2). All other out-links are incident on a different partition and DO NOT form one-step cycles. This property of the index structure is used to detect cycles when deterministic evaluation is requested.

In addition to the edge state, we must also track some state for each vertex. This can also include whether or not the vertex is scheduled for execution or other metadata about the vertex. The VertexState and Scheduled columns have made up values in this example (they were not given in the GraphChi paper).

```
       partition[0]      |      partition[1]       |      partition[2]       |
-------------------------+-------------------------+-------------------------+
 O VertexState Scheduled | O VertexState Scheduled | O VertexState Scheduled |
-------------------------+-------------------------+-------------------------+
 1     1.0          Y    | 3     2.0          Y    | 5     0.0          N    |
 2     1.1          Y    | 4     1.1          N    | 6     4.0          Y    |
```

Note: If we permit a vertex $v$ to update the state (including the schedule) of any vertex whose in-edge or out-edge lies within the one-hop neighborhood of $v$, then it is possible for vertices in other partitions to be updated when $v$ is updated - this would create a fundamental scaling limit and data race for per-vertex state. While it is clearly true that NOT ALL vertices will be updated, we lack any method to group vertex state in a manner to limit the IO that we must perform. Thus, if we allow per-vertex state read and/or write for vertices that are not $O$ vertices in the current memory shard, then we must read and/or write the state of ALL vertices in the graph (or perform random updates against a vertex state index).

We can avoid this scaling limit if we restrict the visible state to just (a) the edges and edge state in the current memory shard; and (b) the vertex state for those vertices that lie at the center of the 1-hop neighborhoods for the current memory shard. (Note that this is more restrictive than the GraphChi API, which permits vertices to be scheduled even when they are the source of an in-edge or the target of an out-edge.) Therefore, it may be better if we schedule edges dynamically, rather than vertices. If any edge is scheduled for a vertex, then that vertex will run.

### *Column Storage*

If we use a column storage model, then, per above, the vertex and edge state can be broken into multiple files (at least one per minor partition). This makes it possible to (a) update the graph computation state without having to rewrite the graph topology; (b) compress the columns for efficient storage; and (c) use variable length encoding for the vertex identifiers, link types, and attribute values. For large graphs, that can be a substantial IO savings, including a reduction in random IO associated with dictionary lookup.

### *Hybrid Database Architecture*

A hybrid of a graph database and GraphChi would be capable of very fast ad hoc query answering and IO efficient graph mining. Further, hybrid vertex program execution strategies could be devised that used the efficient random access indices of the graph database to do much less IO than GraphChi when only a few vertices in the graph are touched by the vertex program. This would require a means to efficiently store updates against the column projections for the edge state and vertex state, but for problems where only a few vertices are touched, the state can be maintained in RAM.

GraphChi provides data-parallel updates within each major partition (with the exception of updates that form one-step cycles within a major partition). If we relax the consistency constraints (GraphChi permits this), then we can compute the entire major partition in a data parallel manner. In combination with GPU support for column compression, this might provide an in-memory layout that was dense and efficient for data parallel operations and that supported IO efficient out-of-core processing.

The proposed approach could also be used to scale-out GraphChi horizontally on a cluster using an existing horizontally scaled graph database such as bigdata® or Accumulo. Both of these platforms provide transparent dynamic key-range partitioning of indices. The additional index required by GraphChi could be built on demand (e.g., by a map/reduce job) or maintained as a dedicated graph-mining index. If major partitions are executed in parallel, then the entire computation could be decomposed onto a cluster. This offers another potential path to a hybrid CPU/GPU graph mining platform with multi-node capability.

## Clustered graph database platforms

### *Index based systems*

#### YARS2 (DERI)

YARS2 (Harth, 2007) is the first clustered graph database. YARS2 used the same pattern of covering indices that was pioneered by YARS (Harth, 2005) and which is used by bigdata among others. YARS2 was designed as a part of the Semantic Web Search Engine (SWSE), an EU project. YARS2 supports quads – the forth position of the quad has come to be know as the named graph and was to indicate the *source* from which the triples were extracted. Covering indices with quads requires six indices: SPOC, POCS, OCSP, CPSO, CSPO, and OSPC where C is the "Context" – also know as the named graph – and is the fourth position of the quad. A batch index build procedure was developed based on merge sort of the indexed quads. The quads were first sorted into the SPOC index order (it would have been more efficient to begin with the CSPO index order since the per-source sort could have been done in parallel for each harvested document). Once the initial index order was complete, the quads were re-sorted into each of other index. As a final step, an inverted Lucene index was built to support full text queries.

YARS2 uses sparse indices. The nodes of the index are retained in RAM and one IO is performed per index block read. (A similar effect is achieved for B+Tree indices by the bigdata platform.) The indices contain the lexical form of the quads, with the common prefix for an index block represented once at the start of that block. Index blocks are compressed using Huffman encoding.

The indices are divided into partitions using the hash code of the first index element is used. E.g., the hash code of S for the SPOC index. Common predicates such as rdf:type can have very high cardinality in web crawls. To avoid exceeding the capabilities of a single machine, and to evenly distribute the data, YARS2 used randomized placement for indices starting with P, e.g., POCS. Other systems using hash partitioning have either hash all triples based solely on the subject (4store) or used the first two key-components for index orders beginning with P. For YARS2, if the index begins with P, then the query is flooded to all nodes since the tuples for the P index are randomly distributed.

YARS2 supports parallel evaluation of concurrent queries. Iterators reading on remote access paths returned sets at a time and supported blocking evaluation in order to avoid overwhelming main memory with intermediate result sets. Indexed nested loop joins were implemented. The joins executed on the query node, with access paths that read on the storage nodes.

### 4store (Garlik/Experian)

4store was developed by Garlik (now part of Experian) to support the company in the development of its possible markets, primarily related to the identification of breaches of personal information. The architecture is based on a hash partitioning of triples into segments and the redundant location of segments on a cluster of commodity hardware nodes. The architecture supports replication, where *r* is the number of replicas. Hence, *r=2* means that there are three copies of a given segment. Writes will fail unless all replicas are online (the cluster must be 100% up for writes to succeed). Reads will succeed (for a given access path) if there is at lease one segment that can answer the request. 4store is designed for relatively small clusters (9 nodes are discussed in the paper) and uses hash codes that offer an expected collision free performance for up to 39B triples. Unspecified mechanisms are used once hash collisions are detected.

In 4store, the data segments are maintained by Storage nodes. Each segment has per-predicate {**P**}S index and a per-predicate {**P**}O index. These indices are tries. In addition, there is an index by "model", which is similar to the concept of a named-graph. The model graph index allows efficient enumeration of all triples in a model and supports efficient delete of the triples for a model. Data partitioning is based on Resource IDentifiers (RIDs). In 4store, a 64-bit RID includes bit flags to mark URIs, Literals, and Blank nodes. The remaining bits are the hash code of the lexical value (for URIs and Literals). For blank nodes, the remaining bits are assigned in a pattern designed to randomize the placement of the blank nodes across the segments. Triples are assigned to segments based solely on the RID of the Subject of the triple. Thus, all triples with a common subject are always present in the same Segment.

Query evaluation is coordinated by a single Processing node. Joins are executed on the Processing node against access paths that read on the Storage nodes (there is an optimization for star-join patterns with a common subject). Join ordering is done based on an analysis of the triple patterns and a statistical summary of the frequency of different predicate types. Load balancing for query is performed by distributing reads to the Storage nodes having a replica of a segment on which an access path must read with the least workload. Access paths for a query may be run in parallel to reduce the total latency of a query, but 4store prefers to run the access paths sequentially. 4store evaluates subject-centric star-joins at the segment for that subject. DISTINCT and REDUCED are pushed down to individual joins to reduce the cardinality of the intermediate solution sets.

4store has been released as open source under a GPL license. Garlik/Experian have internally replaced the use of 4store by the new (and unpublished) 5-store platform.

## Virtuoso (OpenLink)

Virtuoso was originally developed as an RDBMS platform. It has subsequently been extended to support the XML and RDF data models and query languages. Most recently, the platform has been extended into a hybrid of a row-oriented RDBMS and a column store (Erling, 2012). Unlike many RDF databases, Virtuoso does not carry covering indices. Instead, it models graphs using a relational table with S, P, O, and C column and carries multiple indices over that table, providing a variety of access paths (Earling and Mikhailov, 2008; Earling and Mikhailov, 2009). Additional indices may be declared and the query optimizer will use them if they are declared.

Virtuoso makes extensive use of bitmap indices and (most recently) column projections to minimize the on disk footprint associated with RDF data. Virtuoso is also one of the few RDF databases that support query-time inference as opposed to eager materialization of entailments (Earling and Mikhailov, white paper). The most recent release also supports runtime query optimization based on sampling and cutoff joins (Kader, 2009).

Virtuoso is available as open source (GPL) for single machine deployments. The clustered database product is available under commercial licenses and supports two basic partitioning strategies (replication versus static key range partitioning) and failover (which is distinct from replication and relies on redundant services). Only read-committed queries can scale horizontally to replicated nodes. Serializable reads always read against the leader. Thus, the read isolation level is reduced in practice from serializable to read-committed when scaling out the database. The high availability mechanism uses a 2-phase commit and requires all nodes to be available. If a single node is down, then the remaining nodes are still available for read but writes will not be accepted. System administers are required to manage resources and trade off availability and throughput for bulk loads.

## Bigdata (SYSTAP, LLC)

SYSTAP, LLC has developed the open source bigdata® platform since 2006. The platform targets the semantic web database market, but the underlying architecture is inspired by the dynamic horizontal partitioning mechanisms used by the Google bigtable architecture. The database is available as open source (GPLv2) and also resold under a commercial license by a number of channels.

Bigdata has two distinct deployment models – one for a single machine (known as a "journal") and one using dynamic key-range partitioning on clusters (known as a "federation"). The journal mode supports up to 50 billion edges on a single machine and can be replicated as a shared-nothing highly available cluster. The federation mode can support much larger data sets and higher throughput for bulk load and analytic query workloads. The highly available journal mode uses replication to provide horizontal scaling of read rates and supports transparent failover based on a quorum model (both reads and writes are permitted as long as at least a bare majority of the services are synchronized). Bigdata uses Multi-Version Concurrency Control, so readers never block for writers.

The federation uses a dynamic key-range partitioning mechanism. Writes are absorbed onto per-node write-once journals. These journals are pre-sized to 200M each. Once they fill, the old journal is closed and a new journal is opened. The view of each index partition on the old journal is then migrated into a read-only, read-optimized batch build B+Tree file known as an *index segment* file. New writes are absorbed on the new journal, while reads will read through into the old journal(s) and/or index segments. Delete markers prevent reading through to older data once a tuple has been deleted.

The index segment is a highly optimized B+Tree file structure. All nodes of the index segment are in a single contiguous region on the disk, in key order. All leaves of the index segment are in another contiguous region on the disk, and also appear in key order. Since the number of index entries is known in advance, a perfect fit bloom filter is computed during the index segment build and written into the file. When the index segment is opened, the node region and bloom filter are read into memory using sequential IO. Thereafter, there is at most one IO per leaf since the nodes are in memory and the bloom filter can be used to reject IOs for keys that are provably not in the index segment file.

As the index partition absorbs writes, new index segment files are generated and added to the current view of the index partition. Periodically those index segments are compacted into a single index segment file (aka a *compacting merge*). Once the index partition reaches 200M on the disk, a separator key is identified that will split the key-range of the index and the index partition is split into two new index partitions. Index partitions may be relocated to even out hot spots on the cluster. The database remains online for reads and writes during all operations on index partitions. If an index partition has been moved, then the index partition is relocated and the operation is automatically retried. Index partitions are located based on a key or a key-range using a *locator* service. Caching is used to reduce inter-node communications against the locator service and the locator service is not a bottleneck during either load or query operations.

The details of the recycling mechanism depend on the deployment model. For both the journal and the federation, a history retention period establishes a minimum period during which historical commit points will remain available. Light-weight per-transaction read locks ensure that committed state remains available as long as there is a transaction that can read on that commit point. Bigdata provides snapshot isolation for readers based on this simple mechanism. On the Journal, recycling is controlled by the backing store at the level of individual allocation slots and is batched as part of the commit protocol (there is no separate “vacuum process”). In the Federation, recycling proceeds by deleting entire read-only B+Tree segments files and Journal files once they are no longer accessible from the commit points pinned by active transactions. The “write-once” design of the cluster facilitates incremental backup as well as batch recycling. The dynamic partitioning strategy continually optimizes the indices on the disk by absorbing data in write order on the per-node journal and then migrating the data into read order in the segment files.

For the RDF database application layer, there are three indices that provide a dictionary with forward and reverse mappings for lexical values. The forward index (TERM2ID) maps lexical values into 64-bit identifiers. The (ID2TERM) index reverses this mapping. In order to keep down the stride of the dictionary indices, a dedicated index is used for large lexical values. This “BLOBS” index uses the hash code of the lexical value and a collision counter, which provides a small index stride at the expense of more random IOs. The actual lexical items are stored in raw records on the backing store to minimize the stride of the index. In order to avoid hot spots in a federation, the internal identifiers are permuted when they are assigned such that the high order bits vary most quickly. This has the effect of evenly distributing writes across the available index partitions.

The RDF database uses the covering indices strategy described by YARS (Harth, 2007). There are three different operational modes for triples, triples with datum level provenance, and named graphs (quads). Depending on the database mode, there will be either 3 or 6 statement indices (a statement is a triple or a quad, depending on the database mode).

Query evaluation is concurrent at all levels – including the concurrent evaluation of queries, the concurrent evaluation operators within a query plan, and the concurrent evaluation of multiple instances of the same operator. The vectored query engine queues intermediate solutions for each operator, and vectors those solutions into the operator for evaluation. Vectoring is used to reduce the coordination costs and improve IO efficiency through increased locality in the B+Tree, cache, and file system.

Query plans are generated based on static analysis, fast estimates of the cardinality of access paths (based on two key probes into the appropriate index), and a series of rewrite rules designed to handle a variety of optimizations. Real world application queries can involve 50-100 joins. The large state space can break the cost-based optimizers found in most relational database platforms and in platforms like RDF3X. Instead, a number of heuristics are applied to choose among alternative plan structures in a lightweight fashion (ANAPSID, 2011). There are two distinct algorithms for join order optimization. The default is based on the fast cardinality estimates and static analysis of the propagation variable bindings. The other join order optimizer is based on runtime sampling of join graphs as first described in ROX (Kader 2009; Spyros, 2012).

## Analysis

There have not been any good studies of performance across the range of graph databases described here. While a variety of index strategies have been used, most of them index edges. Of the systems described, only diplodocus indexes sub-graphs and it combines this approach with a novel technique for filtering out sub-graphs which cannot join based on ordered lists of the sub-graphs in which each lexical item appears. Indices play a critical role in graph databases, allowing fast selection of sub-sets of vertices, edges, or sub-graphs through an index lookup. Main memory graph processing platforms without indices must do complete scans of all vertices or attributes in order to make equivalent selections. A hybrid architecture is clearly indicated where indices can be used to select relevant parts of a graph for data mining, but data mining itself takes place in main memory – whether in CPU clusters or hybrid CPU/GPU clusters. For example, we might use a declarative query to rapidly subset data at rest in a parallel file system, distributed graph database, and/or relational database. Abstractly, this would look like:

```
RUN { vertex-program }
FROM data-source(s)
SELECT vertex-projection WHERE { graph-pattern+ }
```

### *Comparison*

- 4store defers FILTER evaluation as long as possible in order to avoid the costs associated with the materialization of lexical forms from the dictionary index. Bigdata applies FILTERs as early as possible in order to reduce the number of intermediate solutions into the subsequent access paths and do less work in the following joins. The different approach to FILTER evaluation is mainly due to the query evaluation model. In 4store, the access paths are remote reads and the joins are evaluated on the Processing node. In bigdata, intermediate solutions are routed to the data nodes having the appropriate index partitions and nested index joined are performed at those nodes. Thus, bigdata can benefit from applying filters as early as possible.
- When the predicate is a variable (or a set of predicates are to be visited), star-joins in 4store must scan multiple PS or PO tries in the appropriate segment for the constant S or O. Graph databases having an index starting with S (e.g., SPO) can scan only the key-range for that subject, which provides a more efficient star-join evaluation strategy.
- The simplest join ordering algorithms analyze the number of unbound variables in the triple patterns and make assumptions about how the selectivity of triple patterns change as shared variables become bound. 4store additionally has access to predicate frequency statistics. Bigdata uses two distinct query optimization techniques. One is based on fast estimates of the selectivity of an access path based on two key probes into the corresponding index. The other is based on chain sampling, as first developed by ROX (Kader, 2009). Virtuoso also offers chain sampling in addition to traditional join order optimization techniques.
  Chain sampling identifies the most selective vertices in a *join graph*. Each join graph vertex corresponds to a triple pattern in the original query and is associated with access path. For ROX, that access path must have an index (this is the zero investment principle). A random sample is then taken for those initially selective join graph vertices and fed into each join that leads to another

vertex in the join graph. These joins are *cutoff* – up to N tuples are pushed into the join and evaluation halts as soon as M tuples are output from the join. This provides a runtime estimation of the actual selectivity of the join in the data. These selectivity estimates are used to incrementally expand the query plan. New edges (corresponding to unexplored joins) are added at each cycle, and join paths that are dominated for the same set of join vertices are eliminated. Because it relies on sampling the actual execution of the joins against the data, ROX is robust to correlations in queries and data that can fool other optimizers. ROX always delivers a good join plan. The plan is not guaranteed to be optimal due to sampling, but the sampling makes the plan robust to local regularities or irregularities in the shape of the data. Recently, researchers (Vidal, 2012) have begun to explore the use of symmetric hash joins (Haas, 1999; Gang, 2002) and eddies (Avnur, 2000, Deshpande, 2004) not only to reduce the latency associated with remote access paths, but also to provide a dynamic alternative to static join optimization techniques.

- The use of dictionaries to encode lexical values is common, but not universal, in graph databases. As a counter example, YARS2 directly stores the entire lexical representation of the triple in the index. The upside of this approach is that there are no materialization costs for the dictionary and FILTERs may be evaluated within index operations. The downside is that the stride of the index is significantly increased by the average width of the lexical representation of a triple. RDF3X is another platform that encodes everything in a dictionary. Several key query optimizations used by the RDF3X platform, including how it achieves sideways information passing, rely explicitly on a dictionary encoding of all lexical values as fixed width integers.
- 4store uses hash codes for the internal identifiers to achieve load balancing. Bigdata and Virtuoso apply a rotation to the bits in the internal identifiers in order to have the most quickly varying bits appear in the most significant bits of the identifier. This facilitates load balancing of the indices using those identifiers. Bigdata and Virtuoso also support inlining of data typed literals. By inlining data typed literals, these systems are able to dramatically reduce the size of the dictionary. Values that are typed as xsd:int, xsd:long, xsd:double, xsd:float, xsd:dateTime, etc. appear directly in the statement indices and do not require decoding against a dictionary. Since the value appears in the statement index, key-range scans of the statement indices are possible for strongly typed data and typed triple patterns.

### *Key-Value store and Map/Reduce systems*

There have been several attempts to develop graph databases based on key-value stores such as HBase or Cassandra (e.g., CumulusRDF), on Lucene clusters, and on map/reduce platforms such as Hadoop. Key-value stores provide very little infrastructure for efficient query processing, and it is difficult to create efficient graph databases as applications of these platforms. Several interesting research platforms have been created over Hadoop. Many of these platforms have focused on scalable computation of the RDFS closure for very large graphs (100s of billions of edges). In addition, a few platforms have been developed for answering structured queries against graph data. These systems emphasize scalability and batch processing, but incur very high latency when answering queries due in part to the high startup costs of map/reduce jobs.

### *CumulusRDF (Harth, Ladwig)*

CumulusRDF explores various ways in which a scalable key-value store might be applied to create a scalable graph database capable (in theory) of answering structured graph queries. The authors review several key-value stores and then focus on Apache Cassandra. They compare two different schemes for representing 6 covering indices (ala YARS2) within Cassandra. Specifically, they compare a hierarchical schema that relies on nested supercolumns (a Cassandra specific feature) and a flat representation. Due to data skew with predicate types, special approaches are required for indices whose first key component is P. For example, the hierarchical scheme uses a PO key and leaves the value empty while the flat representation uses a po column name (and leaves the column value empty) because more than one attribute value may occur for a given property and subject. Using a variety of such tricks, the authors are able to create the necessary 6 covering indices. The experiments were limited to a performance evaluation comparing the hierarchical and flat schemas on single access path requests. The flat schema was found to dominate. A limited secondary experiment also characterized the throughput in terms of linked data "gets" based on a simple DESCRIBE execution (all properties for a subject and up to 10k triples whose target is the subject vertex). The paper does not suggest a technique for handling blank nodes, nor a means to execute DESCRIBE requests that subsume blank nodes into the description using the Concise Bounded Description or similar iterative techniques. The results are not directly comparable to other platforms, but do not appear to offer performance at the levels of the dedicated clustered graph database platforms. Analysis is further compounded because the authors are unable to take a top-to-bottom view of the software stack. For example, they attribute some performance differences to design mismatches in Cassandra (many of which are in fact design features for row stores), such as the atomicity of retrieval of a logical row and the corresponding limits on the size of a logical row. These features/limits make dedicated key-value stores unsuitable as a platform for scalable graph database applications. Further, we believe that successful scalable platforms must provide transparency in the software stack and expertise in the architecture and development team in order to realize efficiencies at all layers of the architecture. For example, this was certainly true for the design team for Google's bigtable platform, the first of the key-value stores.

### SHARD

SHARD (Kurt, 2010) is an architecture for SPARQL query answering against flat files stored in HDFS using Hadoop map/reduce jobs to execute joins. The SHARD system was critiqued by (Huang, Abadi, and Ren, 2011), primarily for failing to exploit an efficient RDF storage layer on the nodes and secondarily for failing to take the graph structure into account when devising the data-partitioning scheme and query decomposition. Only small data sets were considered (100s of millions of edges).

### Graph partitioning

Huang, Abadi, and Ren (2011) present an architecture for SPARQL query answering that uses an efficient RDF aware per-node storage layer (RDF3X) and a graph aware partitioning strategy. They present results comparing SHARD (Kurt, 2010), RDF3X on a single machine, hash partitioning by subject (similar to 4store), and both 1-hop and 2-hop undirected graph partitioning. The direct comparison of hash partitioning by subject with graph partition strategies greatly facilitates the interpretation of these results. Edge cuts for high cardinality vertices were explicitly considered, as was the relationship to main memory graph processing systems (e.g., Pregel) (Malewicz 2010) and Bulk Synchronous Parallel (BSP) (Valiant, 1990).

Only small data sets were considered (250M triples on a 20 node cluster) and then only for a single benchmark (LUBM) (Guo, 1990). Only a limited number of the LUBM queries are not Parallelizable without Communication (POWC). Therefore, the experiments provided interesting results only for the hash partitioning and graph partitioning cases and even then, only for LUBM Queries 2 and 9. The architecture optimizes the decomposition of SPARQL queries into POWC queries with the minimum number of non-POWC joins – thus minimizing the number of map/reduce jobs that must be executed. In many ways, the design deliberately compensates for the poor match of map/reduce processing to online structured query answering.

Huang, *et al* demonstrate that partially redundant graph partitioning can provide a significant speed up over hash partitioning. They show that providing 1-hop and 2-hop guarantees can turn many queries into PWOC queries. However, the general applicability of their work is limited by their reliance on the Hadoop map/reduce platform to handle queries that are not POWC. Each map reduce job introduces significant latency into the query – latency that custom-built graph query systems such as 4store, virtuoso, and bigdata do not suffer. With low-latency internode messaging, such as demonstrated by (Weaver and Williams, 2009), better performance may be yielded using different partitioning and parallelization strategies. (Huang considers Weaver and Williams work on computing RDFS closure, but not their work on general SPARQL query answering.) Finally, the graph partitioning and 1-hop and 2-hop guarantees introduce significant latency into the data load procedure.

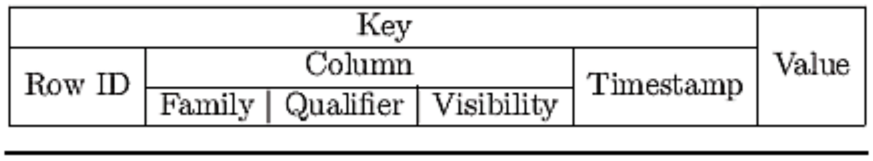
A few other systems have examined the effects of graph partitioning. Sedge (Yang, 2012) provides dynamic partitioning based on workload, including the introduction of redundant partitions in order to minimize internode communications. Significant speedups are reported over Pregel, a system that Huang also considers. While it is not a clustered platform, diplodocus also leverages the graph structure, providing an index of 1-hop sub-graphs. In combination with its template lists (which provide a fast rejection of sub-graphs that can not join) diplodocus also shows significant speedups.

Graph-aware techniques should be closely considered for GPU clusters. However, more research is required to compare methods that rely on efficient interchange of graph data (4store, Urika) or intermediate results (Weaver and Williams, 2009) with methods based on graph partitioning (Huang, 2011; Low 2010) or dynamic graph partitioning (Yang, 2012). Also, unlike GPUs, RDF3X is a disk-based graph storage and query layer. The authors did not consider techniques to distribute or concentrate certain slices of the graphs in order to facilitate parallelization at the cluster level or keep all data related to some aspect of the graph on a single node.

## Accumulo

Apache Accumulo (http://accumulo.apache.org/) was developed in order to expose an architecture having a security model appropriate for DoD applications. Unlike most key-value stores (but in common with bigdata and cassandra), Accumulo maintains the data in key order. Accumulo also supports secondary indices, through experience has shown that the Accumulo indices (private communication) may get out of sync and require occasional rebuild to re-establish coherency.

Accumulo uses the following structure for key-value pairs to model logical rows. The Row ID is the primary key. The Qualifier is the name of the field. Visibility provides security. The Timestamp allows multiple values at different points in time (where the concept of time may be that of the client or the server). The Value is the actual value associated with the rest of the tuple.



| Key | | | | | Value |
|---|---|---|---|---|---|
| Row ID | Column | | | Timestamp | |
| | Family | Qualifier | Visibility | | |

Figure 2 :

The concept of a “Column Family” as originally described for Google’s bigtable is achieved through the concept of “Locality Groups” in Accumulo. These may be changed through an administrative interface. Changes in locality groups are propagated as a side effect on ongoing compacting merge operations.

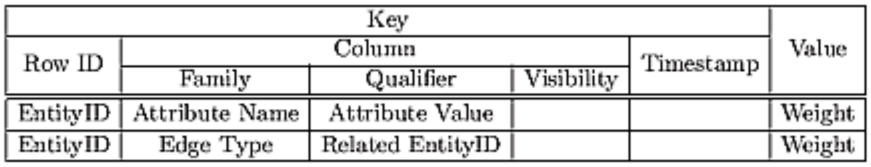
If restated in terms of the Accumulo naming convention, the bigdata key-value store representation looks as follows. In bigdata, the column family is the first component in the tuple and provides the concept of locality groups. Bigdata lacks the concept of Visibility for tuples, though we have considered directly capturing this in the B+Tree tuple data structure.

[ColumnFamily][RowID][ColumnQualifier][Timestamp] : [Value]

Figure 3 :

Accumulo stores the Key and Value as bytes, except for the Timestamp. That is stored as a Long and sorted in reverse order. This reverse sort by timestamp means that the most recent Values appear first in a scan. In bigdata, the entire key is an unsigned byte and logical row retrieval logic forms successor of the desired tuple and then steps backward by one tuple in the index to find the most recent value. This provides the same access speed while maintaining the byte[] generalization for the entire key.

The rationale for the Accumulo tuple design may perhaps be seen when you consider the suggested mapping of graph data into an Accumulo table. Here, the Row ID is the source vertex; the Column Family is appropriated for the edge/property type and the Qualifier for the target vertex. Link weights are optionally associated with the tuple.



| Key | | | | | Value |
|---|---|---|---|---|---|
| Row ID | Column | | | Timestamp | |
| | Family | Qualifier | Visibility | | |
| EntityID | Attribute Name | Attribute Value | | | Weight |
| EntityID | Edge Type | Related EntityID | | | Weight |

Figure 4 :

Thus, Accumulo may be used to represent the edges of a graph directly with both link weights and edge level security. However, in order to provide efficient query answering it is necessary to have multiple orderings over the edges of the graph. This can be achieved through secondary indices. The primary index provides what amounts to an SPO index order with optional link attributes. The secondary indices would provide POS and OSP index orders.

A graph modeled in Accumulo typically uses the external representation of the vertex identifier (Row ID and Qualifier) and the link or property type (Family). This approach is similar to YARS2. The advantage of this approach is the property set for vertex (all tuples having the same Row ID) may be recovered using a single key-range iterator without having to decode internal identifiers by materializing them against a dictionary index. The disadvantages of this approach are (a) an increased footprint on the disk (which can be offset by block compression); (b) the larger index stride would negatively impact performance on joins (again, partly offset through compression); and (c) the in-memory representation of intermediate solutions will be both fatter and slower as joins must be computed against the external representations rather than internal identifiers, which are often simple integers.

Accumulo partitions tablets at the logical row. Thus, the graph scheme proposed above would permit logical rows for the same source vertex to be partitioned across distinct tablets, which could result in a scan crossing a tablet boundary when recovering the metadata for a given entity. Bigdata allows extensible partitioning rules and is able to enforce the constraint that all logical rows for a given source vertex are in the same key-range partition for the index. This constraint can simplify some operations, such as performing star joins at the tablet.

In general, it would be worthwhile to explore the extent to which joins could be distributed in an Accumulo cluster, either by running them on the tablet servers or by running them on a collection of coordinated clients. This is necessary in order to handle high volume queries against very large graphs. For example, coordinated clients would make it possible to use parallel hash joins to avoid client bottlenecks (the data are streamed to multiple clients in parallel from the Accumulo tables) and machine limits (each client sees 1/nth of the total data for the join).

Query consistency could be achieved in Accumulo through a combination of the per-tuple timestamps and a history purge policy. Rather than having snapshot isolation, access paths in Accumulo would provide the datum whose timestamp was LTE to the timestamp associated with the query. This would provide the most current value while ignoring values that had been updated since the time that the query was issues (or since the point in time against which the query was being evaluated). If this level of isolation is not desired, then the query could just run against the most recent datum for a given local row (this is the default behavior for an Accumulo iterator).

Accumulo supports bloom filters. Bloom filters have a space overhead of approximately one byte per tuple. Thus, bloom filters are not practical for scale-up indices or for hash partitioned indices (unless the indices are broken up into multiple partitions on each node). However, perfect fit bloom filters work nicely with the dynamic compaction model used by Accumulo, bigtable, and bigdata. These bloom filters can be leveraged during query evaluation to accelerate vectored nested index joins. This is done by sending a set of key probes to the key-range shard in parallel and receiving back a collection of those probes that were satisfied against the shard. In this situation, the bloom filter leads to significant speedups by correctly reject keys that will not be found in the shard. In bigdata, pipelined nested joins flow across the data server nodes reading local data. However, Accumulo does not appear to expose the facilities require to ship join computation to the tablet servers.

### Accumulo plus Ad Hoc Query (RYA)

RYA (Punnoose, 2012) is a research prototype for ad hoc graph query over Accumulo. It uses multiple index orders (SPO, OSP, and POS) and supports a variety of joins designed to accelerate query answering. The approach is very much like the one outlined above.

Data load is achieved using a map/reduce job (this is the Accumulo bulk load strategy). The batch load time on a 10 node Accumulo cluster is reported as 700 minutes for the LUBM U8000 data set (1 billion edges). Bigdata loads the same data set in one hour on a comparable cluster – nearly 12 times faster. This performance difference is quite interesting as there are known optimizations that would provide substantial further improvement in the bulk load rate for bigdata.

Query evaluation is based on the openrdf platform. Rya uses a map/reduce job to compute statistics that are used to reorder joins. Rya also includes an optimization for parallel evaluation of the nested index subquery joins (shipping an Accumulo iterator with a filter that verifies whether solutions exist for a triple pattern with bound values from the previous join(s)). Finally, an optimization is explored where the Accumulo batch scanner is used to coalesce reads against the same tablets into a common request, which could dramatically reduce the #of actual requests issues (bigdata optimizes this by flowing the intermediate solutions to the tablets and performing the joins at the tablets).

The query results reported by Rya are fatally flawed. The benchmark used (LUBM) does not parameterize the queries in the data. Thus, each presentation of a query presents exactly the same query. Further, the high latency LUBM queries (2,6,9, and 14) were modified to request only the first 100 results. This makes it impossible to compare the performance results reported for Rya with the published results for any other platform. Rya reports low latency for those queries, but this latency is completely artificial as Rya is not executing the full query.

Rya has not examine whether it is possible to parallelize joins (distributed hash joins) or whether it is possible to map the vector the intermediate solutions across the Accumulo tablet servers. High performance for ad hoc query on Accumulo likely depends on the exploration of such performance enhancements.

## *Main memory systems*

Main memory graph databases have very different design characteristics. These databases often lack transactions, provide a single view of a graph, and only support durability through a snapshot of the graph to disk.

### Urika (Cray XMT)

The Cray XMT is a hybrid architecture using Linux nodes with standard CPUs combined with a set of relatively slow stream processing cores capable of context switching between work queues in each clock cycle. In this regard, the XMT is somewhat similar to a hybrid CPU/GPU architecture, but using custom hardware. The XMT also supports unified memory model, which is not true for CPU/GPU clusters. Rather than attempt to optimize the location of data, all memory allocations on the XMT are hash partitioned across the nodes in the cluster and the XMT relies on the interconnect architecture to migrate data to the stream processors. By keeping the stream processors busy, the XMT achieves high throughput on irregular graph structured workloads. Many XMT users directly write low-level applications in a vendor specific parallel extension to C++. Recently, Cray has announced the "Urika" graph appliance. Urika offers an RDF/SPARQL interface to the XMT.

### Scalable RDF query processing on clusters and supercomputers (Rensselaer Polytechnic)

(Weaver and Williams, 2009) developed an unnamed research platform for distributed RDF data load and query on HPC clusters. Unlike nearly all other systems, their platform was designed purely for main memory. They were able to show linear scaling for data load, which is a bottleneck for many platforms, and good parallelism for query on up to 64 or 128 compute nodes, depending on the data set and the query.

Data load was accomplished by block partitioning the input files among the processing nodes in the cluster, building a node-local dictionary encoding lexical forms into 64-bit integers, and building local covering indices (SPO, OSP, and POS) using those 64-bit identifiers. Triple placement was purely a function of what data block was assigned to what node and the same triple could appear on more than one node (the author's do not address the cardinality issues introduced by this redundancy – presumably duplicates were eliminated during hash joins). Solutions were interchange as externalized representations of RDF triples and decoded into node-local identifiers by the receiver. (Presumably, unknown lexical forms were inserted into tables as they were received.) MPI was used to interchange solution sets.

Weaver and Williams used a distributed parallel hash join for all joins. The join operation was distributed to all nodes. Each node would select all local triples for the access path. These data were then marshaled to the node to which they were hashed by the join key. For a query with N triple patterns, they would use N-1 distributed hash joins. The output of each join was reused as the input to the next join. The final solutions were written to disk by each node. Only support for simple hash joins was described. Optional, filters, and aggregation were left as future work.

This system shows excellence query performance using a simple distributed hash-join and relied on a high speed interconnect fabric. However, only relatively small data sets were examined (10s of millions of triples), especially in comparison to the size of the clusters on which the experiments were conducted. The author's investigated the knee where query performance begins to degrade as the number of processes increased but did not reach any firm conclusions regarding the underlying causes or ways in which scaling might be further improved.

#### *Analysis*

The XMT has several design similarities at a hardware level to the proposed approach. It relies on a combination of relatively slow stream processing cores and traditional CPUs to achieve high throughput on graph processing problems.

Philosophically, and if not only for energy reasons, the XMT is problematic moving forward. Locality is increasingly important for both performance and energy reasons - it's simply not tractable to ignore it. In the long run, locality is the only thing that matters. Cray has built an interesting machine, and one that hides a difficult issue for programmers, but we do not believe that it is the right approach. While they make their platform more accessible by offering lease options, the XMT cannot compete with the price/performance and form factor of a workstation with 3 or 4 GPUs, delivering over 8 billion edges per second on commodity hardware.

Hybrid CPU/many-core combinations are the future of computing. GPUs were evolved by the needs of the gaming industry, and are now driven at the high end by the needs of the super-computer industry. GPUs offer price/performance and energy/performance ratios that are unmatched, but it remains a challenge to write efficient parallel software for these commodity processors.

## Physical Schema and Data Structures

### Column Compression on Graphics Processors

(Wenbin, 2010) demonstrates that multi-layered column compression can be efficient on hybrid GPU architectures. This work examines several different compression schemes, including:

- Fixed length null suppression (NS);
- Variable length null suppression (NSV);
- Dictionary encoding (DICT);
- Bitmap encoding (BITMAP);
- Run Length Encoding (RLE);
- Frame of Reference encoding (FOR);
- Delta encoding (DELTA);
- Separation into multiple columns (SEP); and
- Integer scaling (SCALE).

Column compression generally offers more flexibility than row wise compression since the characteristics of each column may be taken into account and the columns compressed independently. Columns are divided into segments to facilitate update, to facilitate scans without an index, and to further improve compression since the compression technique can be adapted to the data within each segment of each column independently.

Modern column stores typically apply a single encoding scheme a column segment. Performance tends to decrease when multiple compression methods are layered (as demonstrated experimentally with GDB and MonetDB). The authors demonstrate that the encoding schemes above may be efficiently applied by a GPU. Further, they provide a means for planning which compression schemes to apply to a column segment based on some simple summary statistics and illustrate how partial decoding may be used depending on the requirements of the query.

This work demonstrates the feasibility of using GPU compression to turn variable length data (such as the representation of edges with inline attribute values) into data with an even stride that can be easily processed by the GPU. For example, an ordered column segment containing variable length SPO edges represented could be compressed using DICT + RLE + NS for S, BITMAP for P, and DICT + NS for O. Further, the specific encoding schemes can be decided dynamically based on an examination of the summary statistics for each column.

Compression can substantially reduce the IO associated with data structures on the disk. The ability to operate efficiently on compressed representations in memory can substantially increase the scale of a graph that can be represented entirely in core and can reduce memory stalls from translation look aside buffer (TLB) misses. While GPU efficient compression and operation on encoded, or partially encoded representations, could dramatically improve performance, it is vital that the compression schemes remain efficient in terms of the warps and the overall graph traversal primitives. Experimentation will be required to quantify and qualify these tradeoffs, but this is a promising direction for integrating attributed graphs with GPU accelerated graph mining.

### Hybrid Row/Column Store

TBD : Description of B+Tree using sparse upper index with column segments linked to the leaf (Virtuso, Vertica, etc).

## Summary and Future Directions

Structured graph query are a useful technique, and one where performance can be compared to a large and growing body of research on graph databases. Some of the most interesting questions are how to design the indexing system, how to evaluate queries efficiently, how to allocate the vertices among the compute nodes in order to balance load and provide good locality for computation. Recently, a number of systems have been proposed that rely on either static or dynamic graph partitioning (diplodocus, Hadoop+RDF3X, Sedge).

### Identifiers

One open question is whether to assign global identifiers to vertices and attributes or to assign node-local identifiers. The downside of global identifiers is that they are a bottleneck during load. By contrast, Weaver and Williams (2009) showed linear scaling in load performance as a function of cluster size. They accomplished this by encoding the graph vertex identifiers, link names, and attribute values into a per-node dictionary as 64-bit integers. The encoded representation was then used by the node-local data structures. In order to interchange vertex identifiers or attribute values with other compute nodes during query processing, the lexical representation of those data were materialized from the node-local dictionary and then communicated over the interconnect. The receiving nodes would then resolve the lexical forms to internal 64-bit node identifiers – any unknown lexical forms would be assigned new node-local identifiers. The system they describe loaded the data for every query – a cost they could afford since load performance scaled linearly and they dealt with relatively small data sets (5M LUBM triples and 50M Barton triples). However, in an application with larger data sets, the load times would increase shifting the effective cost of load+query operations. Further, the number of lexical forms that were encoded on each node in order to support interchange of intermediate values would also increase with the data set size or if more than one query was performed for a given load operation. These concerns raise questions about the scalability of the per-node dictionary encoding technique.

Unlike relational databases, graph attributes (certainly when interchanged as RDF data) are schema fluid. In practice, this flexibility in the data types is critical to permitting arbitrary data sets to be combined. However, it also means that you cannot rely on strongly typed columns. Effectively, all attribute values are "ANY."

This has implications for data compression techniques. One common practice is to *inline* the representation of values according to their data type. Both bigdata and virtuoso support this, though in slightly different ways. For bigdata, an index on an attribute value will be in key order within each attribute type. For virtuoso, a total ordering is imposed over the different numeric attribute types with their differing typed values appearing at appropriate positions in that total ordering. However, in both cases substantial savings are realized in the indices. The use of GPUs changes the merits of the design tradeoffs here. The inline representations use space that is generally one byte larger than the actual data type. This leads to variable length encoding of attributes. But that variable stride is problematic for GPUs as it can lead to divergence in warps. One approach is to used a fixed with encoding, typically using a 64-bit integer. This allows bit marking of vertex identifiers as URIs, blank nodes, edges (for edge attributes), or specific data type literals and typically leaves 64-bits available to encode the vertex identifier or attribute value.

The disadvantage of this technique is that both magnitude and precision are lost as it is no longer possible to capture IEEE double precision floating point values, 64-bit integers, or xsd:dateTime within a fixed stride. Finally, arbitrary precision floating point (xsd:decimal) and arbitrary magnitude integers (xsd:integer) also cannot be captured in this manner as inline values, requiring either a fallback to dictionary coding or accepting a loss of precision or magnitude. Another way to reconcile this is to use coding techniques from column stores to encode and decode data on the way into and out of the device. This allows the most parsimonious and device efficient encoding to be selected dynamically. In order for the GPU to select specific vertices or interpret encoded attribute values, as is common with constraints, it is also necessary to communicate the encoding strategy to the device. One possibility that could offer the benefits of fast load without the scaling limits of asymptotically complete per-node dictionaries is per-operation encoding at the device. There has been some study of RDF processing on GPUs, notably (Senn, 2010) and (Henio, 2012). These techniques and other mechanisms for encoding rich graphs need to be investigated further before deciding on an approach to rich graph processing on GPU compute clusters.

## Graph partitioning

Researchers on graph database and graph processing platforms have recently begun to investigate the role of graph partitioning in providing scalable performance. Diplodocus uses an online graph partitioning strategy where templates determine how the graph is partitioned into sub-graphs. This system shows excellent performance and a paper on a clustered version of that architecture should be out soon (private communication). Huang (2011) investigated hash partitioning and 1-hop and 2-hop graph partitioning. However, due to the desire to avoid the severely high overhead of managing internode communications using Hadoop, this work focused on Parallelizable without Communication (PWOC) query plans. Those plans effectively limited the response time of the system to the response time of a single node (plans which were executed entirely on a single node were considered to be optimal). Thus, any improvement in performance as a function of data scale came solely through the reduction in the per-node index sizes and the benefit of relatively more working memory for a given query. In contrast, work by Weaver and Williams showed good performance gains as a function of cluster size using parallel hash joins and arbitrary partitioning (Weaver, 2009), but only for small data sets. Again, more research is needed on the intersection of rich graphs, graph partitioning and the data structures for local processing of graphs on the nodes.

## Physical schema and data structures

There are a number of possible data structures for processing large graphs on GPU clusters. These data structures should be evaluated on a variety of graphs and algorithms in order to develop a better appreciation for their applicability to accelerated graph processing on GPUs. The choice of data structure interacts with both the opportunity for compression and the locality of the operations. With respect to data structures for per-link type partitioning versus rich graph representations, whether we use one or the other depends on how much they're accessed together which is partly a function of the problem domain and partly a function of how the problem is decomposed. We should evaluate implementations that would support either model, split or combined.

- Project a sparse matrix for each link type and project columns for attribute values. The advantage here is that the matrix projections are familiar data structures on the GPU and are likely to reduce divergence of threads in warps. This still leaves open the question of how to partition the graph across the GPUs. We would like to compare static placement of vertices, static graph partitioning, and dynamic graph partitioning (Sedge: Yan, 2012).

- Explicitly group the vertices and attributes for sub-graphs. This is basically the approach that diplodocus has taken. "Templates" are used to break the graph into sub-graphs. The sub-graphs are stored as highly compressed records and decompressed on access. The template lists in the dictionary are used to avoid access to sub-graphs in which vertices are known to not appear. There is an open question concerning how to allocate vertices to compute nodes in order to maximize the opportunity for locality across sub-graphs.
- GraphChi demonstrates an IO efficient design for graph mining that can be adapted for graph database architectures. Parallel clustered index scans are used to efficiently "zip" together the 1-hop neighborhood of vertices for computation. Updates are read and written using sequential IOs to maximize the throughput of the disk system.
- Modeling edges using column projections rather than sparse matrices (Heino, 2012). Column stores offer good compression (critical on GPUs) and are optimized for main memory. Recently, some research groups have investigated hybrid CPU/GPU column stores (Boncz, 2006; Pirk, 2012; Pirk, 2012b; Alenka, https://sourceforge.net/projects/alenka/files/ (Apache license)).

## APIs and Cross cutting concerns

There are a number of issues that cut across the different aspects of graph processing in hybrid CPU/GPU architectures that have not been addressed in depth in the other sections of the literature review. Many of these issues relate to the question of the appropriate level for APIs. A list of these issues appears here for completeness.

Vertex programs rely either on bulk synchronous parallel (BSP) or asynchronous (ASP) message patterns. The Pregel platform used BSP. Signal/Collect reported performance gains and improved convergence with ASP. Graphlab begin with BSP and then developed ASP support, including a variety of consistency levels for updates of the temporary data associated with vertex and edge state. Distributed databases tend to approach this problem differently, sending and receiving large blocks of data in order to support efficient operations, but treating the distinction between synchronous and asynchronous communication as one of at-once versus vectored operations and recognizing synchronization points at the levels of queries, operators, and critical points within operators (e.g., the anyway routing technique of eddies). The manner in which messages are queued and interchange for a hybrid CPU/GPU architecture is open question, as are the roles of the CPU and the GPU in interchange. This question interacts with design decisions such as the choice of the partitioning strategy and whether to use global or local encoding of lexical values of vertex identifiers and attribute values.

Many graph-processing platforms allow arbitrary serializable messages to be interchanged. GPU computing platforms tend to exchange C data structures using MPI. It is possible that generic data structures such as protocol buffers might be used to facilitate operations not only within a hybrid CPU/GPU platform, but also with extensions into other application platforms. However, protocol buffers are not always a good solution and fat messages should be discouraged in the memory-constrained environment of the GPU.

A related question is the level at which users will write their algorithms. For the moment, we are assuming that all kernels are written in CUDA and will run natively on the device. Obviously, this provides the greatest flexibility at the expensive of requiring considerable user level knowledge of the data structures and the details of correct and efficient coding for the GPU. Vertex programming has been suggested as a universal API based on either synchronous or asynchronous message passing. However, it is not yet clear whether vertex programming is a sufficiently robust model to handle general problems in approximate graph search and graph pattern mining (Xifeng Yan, private communication). Equally, while the Signal/Collect platform has demonstrated that it is possible to compile structured graph pattern matching queries into vertex programs, it is an open question whether this approach can compete with the cache conscious techniques of main memory databases, including a wide range of data representations and join algorithms.

An alternative approach is to provide declarative languages that allow computation to be decoupled with implementation specific data structures and algorithms. Databases have traditionally followed this approach, allowing a distinction between the logical and physical schema and greatly facilitating optimization. Whether suitable declarative interfaces could be defined for graph processing is an open question.

## References

Andreas Harth, Jürgen Umbrich, Aidan Hogan, Stefan Decker: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. ISWC/ASWC 2007: 211-224

Anicic, Darko, et al. "Ep-sparql: a unified language for event processing and stream reasoning." *Proceedings of the 20th international conference on World wide web*. ACM, 2011.

Auer, Sören, et al. "Dbpedia: A nucleus for a web of open data." *The Semantic Web* (2007): 722-735.

Bancilhon, Francois, et al. "Magic sets and other strange ways to implement logic programs." *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 1985.

Binna, Robert, et al. "SpiderStore: Exploiting Main Memory for Efficient RDF Graph Representation and Fast Querying." *Proceedings of the Workshop on Semantic Data Management (SemData) at VLDB*. 2010.

Biron, Paul, Ashok Malhotra, and World Wide Web Consortium. "XML schema part 2: Datatypes." *World Wide Web Consortium Recommendation REC-xmlschema-2-20041028* (2004).

Bolles, Andre, Marco Grawunder, and Jonas Jacobi. "Streaming SPARQL-extending SPARQL to process data streams." *The Semantic Web: Research and Applications*

(2008): 448-462.

Boncz, Peter, et al. "MonetDB/XQuery: a fast XQuery processor powered by a relational engine." *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006.

Cohen, M. S., & Thompson, B. B. (2001). Training teams to take initiative: Critical thinking in novel situations. In E. Salas (Ed.), Advances in Human Performance and Cognitive Engineering Research (Vol. 1, pp. 251-91). Amsterdam: JAI. Available in http://www.cog_tech.com/papers/CriticalThinking/Training%20Critical%20Thinking%20for%20Initiative%20HFES%202002.pdf

Cohen, M. S., & Thompson, B. B. (2005). Metacognitive processes for uncertainty handling: Connectionist implementation of a cognitive model. In M. Anderson & T. Oates (Eds.), Metacognition in Computation: Papers from the 2005 Symposium (pp. 36-41). Menlo Park, CA: American Association of Artificial Intelligence.

Della Valle, Emanuele, et al. "A first step towards stream reasoning." *Future Internet–FIS 2008* (2009): 72-81.

Deshpande, Amol. "An initial study of overheads of eddies." *ACM SIGMOD Record* 33.1 (2004): 44-49.

Erling, O. DEBULL, I. *(ed.)* Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.,* **2012**, *35*, 3-8

Erling, Orri, and Ivan Mikhailov. "RDF Support in the Virtuoso DBMS." *Networked Knowledge-Networked Media* (2009): 7-24.

Erling, Orri, and Ivan Mikhailov. "SPARQL and Scalable Inference on Demand." (http://www.openlinksw.co.uk/virtuoso/Whitepapers/pdf/Scalable_Inference.pdf)

Erling, Orri, and Ivan Mikhailov. "Towards web scale RDF." *4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2008)*. 2008.

Erling, Orri. "Virtuoso and Database Scalability." (http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSScale).

Guo, Yuanbo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems." *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2 (2005): 158-182.

Harris, Steve, Nick Lamb, and Nigel Shadbolt. "4store: The design and implementation of a clustered RDF store." *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. 2009.

Harth, Andreas, and Stefan Decker. "Optimized index structures for querying rdf from the web." *Web Congress, 2005. LA-WEB 2005. Third Latin American*. IEEE, 2005.

Heino, Norman, and Jeff Z. Pan. "RDFS Reasoning on Massively Parallel Hardware." IDWC, 2012.

Jacopo Urbani, Jason Maassen, and Henri Bal. 2010. Massive Semantic Web data compression with MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (HPDC '10). ACM, New York, NY, USA, 795-802. DOI=10.1145/1851476.1851591 http://doi.acm.org/10.1145/1851476.1851591

Kyrola, Aapo, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-scale graph computation on just a PC." OSDI, 2012.

Ladwig, Günter, and Andreas Harth. "CumulusRDF: Linked data management on nested key-value stores."

Lassila, Ora, and Ralph R. Swick. "Resource description framework (RDF) model and syntax specification." (1998).

Lindberg, Donald A., Betsy L. Humphreys, and Alexa T. McCray. "The Unified Medical Language System." *Methods of information in medicine* 32.4 (1993): 281.

Low, Yucheng, et al. "Graphlab: A new framework for parallel machine learning." *arXiv preprint arXiv:1006.4990* (2010).

Luo, Gang, et al. "A scalable hash ripple join algorithm." *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002.

Malewicz, Grzegorz, et al. "Pregel: a system for large-scale graph processing." *Proceedings of the 2010 international conference on Management of data*. ACM, 2010.

Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. 2011. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *Proceedings of the 10th international conference on The semantic web - Volume Part I* (ISWC'11), Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, and Abraham Bernstein (Eds.), Vol. Part I. Springer-Verlag, Berlin, Heidelberg, 18-34.

McCray, Alexa T., and Stuart J. Nelson. "The representation of meaning in the UMLS." *Methods of information in medicine* 34.1-2 (1995): 193.

Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, Bhavani Thuraisingham: Storage and Retrieval of Large RDF Graphs Using Hadoop and MapReduce, CloudCom 2009: 680-686.

Open Geospatial Consortium. "OGC GeoSPARQL-A geographic query language for RDF data." *Open Geospatial Consortium* (2011).

Patchigolla, Venkata. "Comparison of clustered RDF data stores." (2011).

Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple joins for online aggregation. *SIGMOD Rec.* 28, 2 (June 1999), 287-298. DOI=10.1145/304181.304208 http://doi.acm.org/10.1145/304181.304208

Pirk, Holger, et al. "X-device query processing by bitwise distribution." *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 2012.

Pirk, Holger. "Efficient Cross-Device Query Processing." *Proceedings of the International Conference on Very Large Data Bases. Istanbul, Turkey. VLDB Endowment*. 2012.

Prud'Hommeaux, Eric, and Andy Seaborne. "SPARQL query language for RDF." *W3C recommendation* 15 (2008).

Przyjaciel-Zablocki, Martin, et al. "RDFPath: path query processing on large RDF graphs with mapreduce." *The Semantic Web: ESWC 2011 Workshops*. Springer Berlin/Heidelberg, 2012.

R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen, "ROX: Run-Time Optimization of XQueries," in SIGMOD, 2009.

Rohloff, Kurt, and Richard E. Schantz. "High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store." *Programming Support Innovations for Emerging Distributed Applications*. ACM, 2010.

Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: continuously adaptive query processing. *SIGMOD Rec.* 29, 2 (May 2000), 261-272. DOI=10.1145/335191.335420 http://doi.acm.org/10.1145/335191.335420

Roshan Punnoose, Adina Crainiceanu, and David Rapp. 2012. Rya: a scalable RDF triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence* (Cloud-I '12). ACM, New York, NY, USA, , Article 4 , 8 pages. DOI=10.1145/2347673.2347677 http://doi.acm.org/10.1145/2347673.2347677

Schätzle, Alexander, Martin Przyjaciel-Zablocki, and Georg Lausen. "PigSPARQL: Mapping SPARQL to Pig Latin." *Proceedings of the International Workshop on Semantic Web Information Management*. ACM, 2011.

Senn, Juerg. "Parallel Join Processing on Graphics Processors for the Resource Description Framework." *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*. VDE, 2010.

Spyros Kotoulas, Jacopo Urbani, Peter Boncz, Peter Mika, "Robust Runtime Optimization and Skew-Resistant Execution of Analytical Queries on Pig" in ISWC 2012.

Theobald, Martin, et al. *URDF: Efficient reasoning in uncertain RDF knowledge bases with soft and hard rules*. Tech. Rep. MPI-I-2010-5-002, Max Planck Institute Informatics (MPI-INF), 2010.

Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. The VLDB Journal 19, 1 (February 2010), 91-113. DOI=10.1007/s00778-009-0165-y http://dx.doi.org/10.1007/s00778-009-0165-y.

Valiant, Leslie G. "A bridging model for parallel computation." *Communications of the ACM* 33.8 (1990): 103-111.

Vidal, María-Esther, et al. "Efficiently joining group patterns in SPARQL queries." *The Semantic Web: Research and Applications* (2010): 228-242.

Weaver, Jesse, and Gregory Todd Williams. "Scalable RDF query processing on clusters and supercomputers." *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. 2009.

Weibel, Stuart, et al. "Dublin core metadata for resource discovery." *Internet Engineering Task Force RFC* 2413 (1998): 222.

Wick, M., and B. Vatant. "The geonames geographical database." *Available from World Wide Web: http://geonames. org*.

Wolf, Joel L., Daniel M. Dias, and Philip S. Yu. "An effective algorithm for parallelizing sort merge joins in the presence of data skew." *Databases in Parallel and Distributed Systems, 1990, Proceedings. Second International Symposium on*. IEEE, 1990.

Wylot, Marcin, et al. "dipLODocus RDF _ RDF—Short and Long-Tail RDF Analytics for Massive Webs of Data." *The Semantic Web–ISWC 2011* (2011): 778-793.

Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proc. VLDB Endow.* 3, 1-2 (September 2010), 670-680.

Yang, Shengqi, et al. "Towards effective partition management for large graphs." *Proceedings of the 2012 international conference on Management of Data*. ACM,

|  |
|---|
| 2012. |